Kotlin开发
快速入门与实战

王志强 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书主要介绍在 Kotlin 开发中各种相关的技术及知识，全书共分为 7 章，内容层次清晰，难度循序渐进。第 1 章介绍 Kotlin 编程环境的搭建、如何运行 Kotlin 项目，以及编辑器的安装；第 2 章介绍 Kotlin 编程语言的基础，主要内容包括变量与常量、常见的数据类型、运算符以及流程控制语句；第 3 章介绍集合，以及常见的集合操作；第 4 章介绍函数和函数的使用；第 5 章主要介绍面向对象基础知识和高级编程；第 6 章主要介绍 Kotlin 与 Java 的互操作、Kotlin 与 JavaScript 的互操作；第 7 章主要介绍一个实例——电子拍卖系统，通过 Kotlin 编写 Android 客户端。

如果你有一定的 Java 编程基础，则可以很容易理解 Kotlin 编程。没有 Java 编程基础也没有关系，本书也是从 Kotlin 编程基础开始讲起的。希望通过阅读本书，能够让你成为一个全栈工程师。

图书在版编目（CIP）数据

Kotlin 开发快速入门与实战 / 王志强著. —北京：电子工业出版社，2017.10
ISBN 978-7-121-32517-5

Ⅰ. ①K… Ⅱ. ①王… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2017）第 199386 号

策划编辑：黄爱萍
责任编辑：葛 娜
印 刷：
装 订：
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编：100036
开 本：720×1000 1/16 印张：13.25 字数：237 千字
版 次：2017 年 10 月第 1 版
印 次：2017 年 10 月第 1 次印刷
定 价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

前　言

很多人想学习 Kotlin 这门编程语言，却不知道该如何下手；有些人懂得 Java 和 Android 的基本语法，却不知该如何使用 Kotlin 进行应用程序的开发；本书就是为这些人准备的学习与开发指南。正所谓知识来源于实践，实践是检验真理的唯一标准，本书严格遵守这一原则，对每一个知识点都进行了案例分析，帮助读者真正掌握和运用 Kotlin。

为什么要读这本书

如果你不知道这本书是否能帮助到自己，或者不知道是否要选择这本书，那么请先想一想在平时的学习或工作中是否遇到过以下这些问题：

- 有想用 Kotlin 开发 APP 的想法，但是却不知道该如何下手；
- 刚学习了编程语言的 if、for、while 等各种语法，却不知道利用它们到底能做些什么；
- 精通 CPP、Java 等编程语言，却不知道如何配合 Kotlin 开发新的技术。

如果上述问题是你正在困惑的，那么在你不知所措时或许本书能帮助到你。阅读本书能帮助你解决工作中的一些实际问题！

本书特色

1．零基础

在学习本书之前不需要具备任何的计算机专业背景，任何有志于 APP 开发的读者都能利用本书从头学起。本书在基础知识和实践部分都有大量案例，代码简短而精湛，紧扣知识点的本质，以加深印象；同时结合作者多年的项目开发经验，

阐述了很多代码编写技巧，读者可以将代码复制到自己的计算机上自行实践和演练。本书相关案例代码可以通过添加 QQ 群：99208965，自行下载。

2．合理的章节安排

本书首先讲解了 Kotlin 语言的基础知识和编程风格等内容，然后详细介绍了 Kotlin 的互操作，最后通过项目实战帮助读者综合运用所有的知识点。

3．典型的项目案例

作者根据多年的项目经验，将典型的案例与知识点相互整合，方便读者理解、巩固每章的知识点。最后一章介绍的项目案例不仅可以让读者在实际应用中更加熟练地掌握前面讲到的知识点，更能让读者了解在 Kotlin 开发应用程序中从轮廓到细节的完整实现流程。

内容安排

本书分为 7 章，内容覆盖 Kotlin 编程基础知识和项目开发实战。

第 1~4 章系统介绍 Kotlin 语言，并且阐述应用程序开发必备的基础知识。这些内容不仅适合新手学习，对有经验的开发者同样适用。

第 5~6 章系统介绍 Kotlin 编程中的面向对象知识，并且详细讲解面向对象的三大特性，以及在 Kotlin 中常见的类。

第 7 章通过电子拍卖系统，详细讲解如何使用 Kotlin 编程语言进行 Android 开发，并使用 PHP 脚本语言与 Android 客户端进行数据交互，使得读者能够深入浅出地学习和实践，并努力成为全栈开发工程师。

作者按照自己的开发经验编排了本书的章节顺序，推荐读者也按顺序阅读，尤其不能跳过第 1~4 章介绍的基础知识。如果读者的阅读时间特别紧迫，也可以在阅读完第 1~4 章后，直接阅读所需要的部分内容。

本书配套源码下载地址：https://github.com/cnkotlin。

读者对象

- IT 技术爱好者

- Android 开发工程师
- 全栈开发工程师
- 大中专院校及各 IT 培训学校的教师与学生
- 希望自己能够独立实现 APP 开发的程序员

致谢

首先要特别感谢家人，感谢您们的理解和鼓励。其次要特别感谢在背后默默支持我的朋友们，若没有你们的支持和鼓励就不会有此书的出版，能够遇到这些聪明、经验丰富、趣味相投的朋友是人生一大幸事。

还要特别感谢电子工业出版社的黄爱萍和葛娜编辑，感谢你们一直在背后给予我的支持和鼓励，以及在策划和稿件整理方面做出的大量工作。

王志强
惠铭科技联合创始人兼 CTO
2017 年 8 月 1 日

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书所提供的源码文件，可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32517

目　录

第1章 Kotlin环境搭建和开发工具

Kotlin是基于JVM的编程语言，由JetBrains公司研发和维护，可以将它编译成Java字节码，也可以编译成JavaScrpit，方便在没有JVM的设备上运行。

Kotlin 的优点在于简洁、开源、容易使用、安全以及工具支持，其中最大的优势是可以和其他编程语言（如Java、JavaScript等）进行交互。

要使用Kotlin，首先需要建立Kotlin开发环境。本章主要介绍在Windows和Linux 操作系统下进行 Kotlin 环境的搭建，以及流行开发工具的使用。此外，还将介绍使用Kotlin官方提供的编辑器编写第一个Kotlin示例。

本章主要内容：

- 在Windows操作系统下配置Kotlin环境
- 在Linux操作系统下配置Kotlin环境
- 常见的Kotlin开发工具
- 第一个Kotlin示例

1.1 在Windows操作系统下配置Kotlin环境

在Windows操作系统下安装Kotlin，流程如下：

（1）下载JDK。

（2）安装JDK。

（3）配置JDK环境变量。

（4）检查JDK是否安装。

（5）下载Kotlin编辑器。

（6）配置Kotlin环境变量。

（7）检查 Kotlin 是否安装成功。

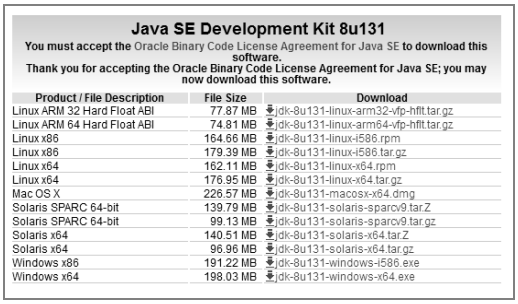
下面具体讲述 Kotlin 环境的配置。

1.1.1 在 Windows 操作系统下安装及配置 JDK

JDK 是 Java 语言的软件开发工具包，由于 Kotlin 与 Java 语法太相近，在 Kotlin 中无时无刻不在与 Java 相关内容打交道，所以，在安装 Kotlin 之前要先配置 JDK 环境。

（1）进入 Oracle 官方网站（http://www.oracle.com），找到 Java SE Development Kit 并进行下载，如图 1-1 所示，根据不同的操作系统选择不同的版本。

（2）下载完成之后，将得到一个 JDK 安装包，如图 1-2 所示。



Java SE Development Kit 8u131

You must accept the Oracle Binary Code License Agreement for Java SE **to download this software.**

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.87 MB	jdk-8u131-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.81 MB	jdk-8u131-linux-arm64-vfp-hflt.tar.gz
Linux x86	164.66 MB	jdk-8u131-linux-i586.rpm
Linux x86	179.39 MB	jdk-8u131-linux-i586.tar.gz
Linux x64	162.11 MB	jdk-8u131-linux-x64.rpm
Linux x64	176.95 MB	jdk-8u131-linux-x64.tar.gz
Mac OS X	226.57 MB	jdk-8u131-macosx-x64.dmg
Solaris SPARC 64-bit	139.79 MB	jdk-8u131-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.13 MB	jdk-8u131-solaris-sparcv9.tar.gz
Solaris x64	140.51 MB	jdk-8u131-solaris-x64.tar.Z
Solaris x64	96.96 MB	jdk-8u131-solaris-x64.tar.gz
Windows x86	191.22 MB	jdk-8u131-windows-i586.exe
Windows x64	198.03 MB	jdk-8u131-windows-x64.exe

图 1-1

图 1-2

（3）双击 JDK 安装包，会出现如图 1-3 所示的安装界面。

图 1-3

（4）单击“下一步”按钮，会出现如图 1-4 所示的定制安装界面。

图 1-4

（5）单击“下一步”按钮，就会出现如图 1-5 所示的安装进度界面。

图 1-5

（6）耐心等待，随后会出现如图 1-6 所示的“Java 安装-目标文件夹”界面，这时要安装的是语言支持插件，直接单击“下一步”按钮即可。

图 1-6

（7）安装进度达到 100%之后，会出现安装完成界面，如图 1-7 所示。单击“关闭”按钮，即可完成 JDK 的安装。

图 1-7

接下来是关键的时刻——配置系统环境变量。首先应了解要配置的系统环境变量，分别是 Java_Home、Path 和 ClassPath。

（8）选择桌面上的“计算机”图标并单击鼠标右键，在弹出的快捷菜单中选择“属性”→“高级系统设置”→“高级”→“环境变量”，打开如图 1-8 所示的“环境变量”对话框。

图 1-8

小知识

系统变量：与 Windows 操作系统包括网络状况有关，由操作系统定义。Administrators 组的用户可以添加、修改、删除系统变量。

用户变量：由操作系统、某些应用程序和用户建立，例如 Windows XP 安装程序将临时文件夹设置为默认存储位置，并视为用户变量。任何用户都可以添加、修改、删除用户变量。

（9）单击“新建”按钮，打开如图 1-9 所示的“新建系统变量”对话框，在“变量名”框中输入“Java_Home”，在“变量值”框中输入“C:\Kotlin\Java\jdk18”（根据自己的安装目录来填写），然后单击“确定”按钮。

图 1-9

（10）接下来配置 ClassPath。首先要查看在系统变量中是否有 ClassPath 项目，如果没有则单击“新建”按钮；如果已经存在 ClassPath 项目，则单击“编辑”按钮，打开如图 1-10 所示的“编辑系统变量”对话框，在“变量值”框中填写“C:\Kotlin\Java\jre18\lib”（根据自己的安装目录来填写），填写完成后，单击“确定”按钮。

（11）接着进行 Path 的配置。同上述 ClassPath 的配置，在“变量值”框中填

写“C:\Kotlin\Java\jdk18\bin”（根据自己的安装目录来填写），然后单击“确定”按钮，如图 1-11 所示。

图 1-10

图 1-11

至此，JDK 配置完成。接下来检查 JDK 是否配置成功。

（12）同时按下“Windows+R”快捷键，将打开如图 1-12 所示的“运行”对话框。

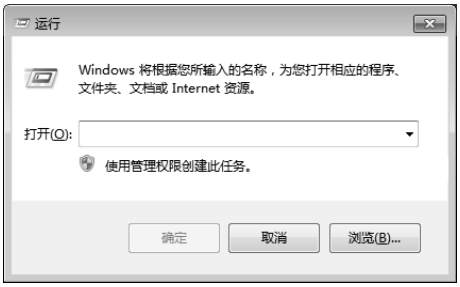


图 1-12

（13）在“打开”后面的输入框中输入“cmd”后，单击“确定”按钮，将出现如图 1-13 所示的 DOS 命令行窗口。

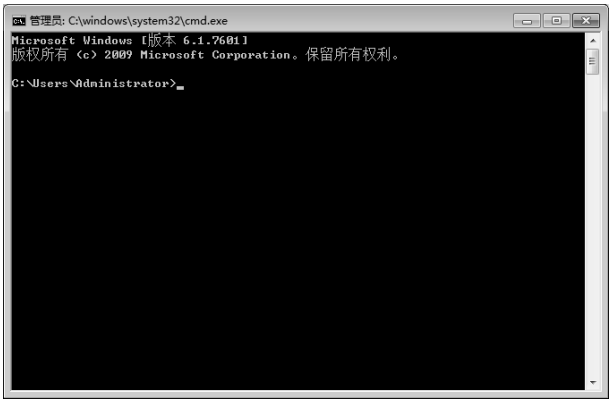


图 1-13

（14）在“命令提示符”后输入“java -version”，可以查看 JDK 的版本信息；输入“javac”，将会出现对应命令的帮助信息，如图 1-14 所示。

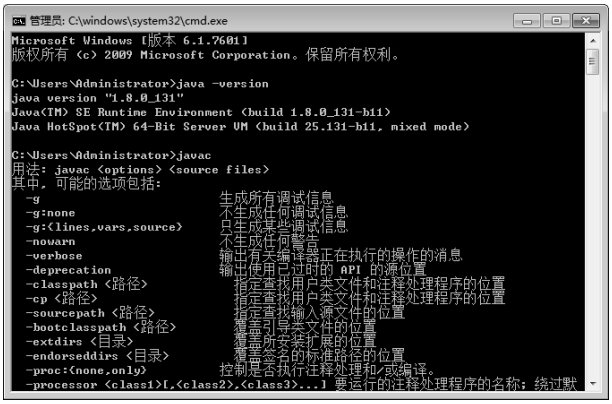


图 1-14

1.1.2　在 Windows 操作系统下安装及配置 Kotlin

配置好 JDK 环境后，下面进行 Kotlin 的安装及配置。

（1）进入 Kotlin 官方网站（http://kotlinlang.org），下载独立的编辑器，如图 1-15 所示。

（2）下载完成之后，将得到一个压缩包，如图 1-16 所示。

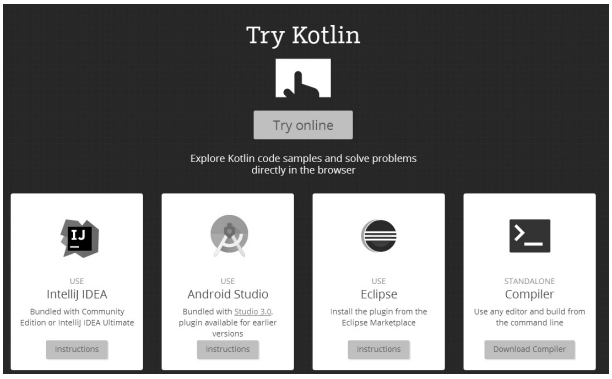


图 1-15

图 1-16

（3）将其解压缩到 C:\Kotlin\Kotlin\目录下，将得到如图 1-17 所示的文件夹和文件。

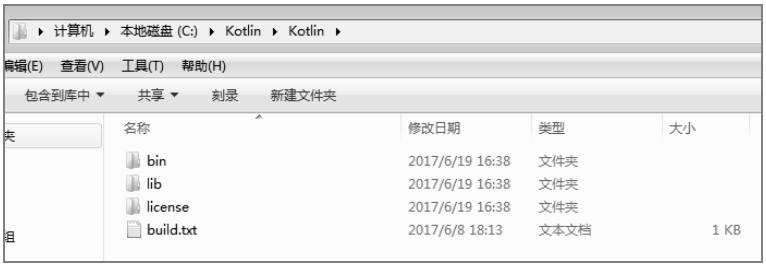


图 1-17

接下来开始配置 Kotlin 环境变量。与 1.1.1 节配置 JDK 环境变量的过程类似，需要添加（或编辑）变量“Kotlin_HOME”和“Path”。

（4）添加“Kotlin_HOME”变量，配置如图 1-18 所示。

（5）编辑“Path”变量，配置如图 1-19 所示。

图 1-18

图 1-19

（6）在“命令提示符”后输入“kotlin -version”，可以查看 Kotlin 的版本信息，如图 1-20 所示

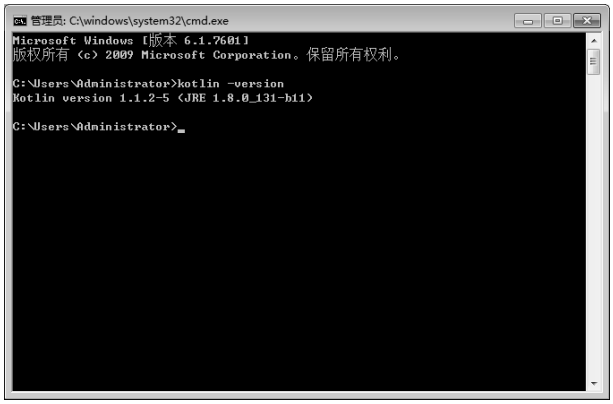


图 1-20

至此，在 Windows 操作系统下成功配置完成 Kotlin 环境。

1.2　在 Linux 操作系统下配置 Kotlin 环境

本节讲解在 Linux 操作系统下安装及配置 Kotlin。另外，在 Mac OS 系统下安装及配置 Kotlin 的过程，与在 Linux 系统下进行安装及配置比较相似，使用 Mac OS 系统的读者可以参考本节内容进行相应的操作。

这里选择的 Linux 操作系统为 CentOS 6.6，64 位。

小知识

常见的 Linux 发行版有 Debian、Gentoo、Ubuntu、Damn Vulnerable Linux、CentOS、Fedora、Kali Linux、Arch Linux、openSUSE 等。

常见的 Linux 服务器有 Debian、Ubuntu、SUSE、openSUSE、FreeBSD、CoreOS 等。

1.2.1　在 Linux 操作系统下安装及配置 JDK

在 Linux 操作系统下安装及配置 Kotlin 之前，依然要先安装及配置 JDK 环境。

（1）下载 JDK 完成之后，将得到一个压缩包，如图 1-21 所示。

（2）使用 SecureCRT 远程连接工具登录服务器，使用 rz 命令把该压缩包上传到 Java 目录下（如果在使用 rz 命令时提示“-bash: rz: command not found”，则使用“yum install -y lrzsz”命令进行安装，安装之后方可使用 rz 命令），再使用 ls 命令查看压缩包 jdk-8u131-linux-x64.tar.gz 是否成功上传到服务器中。结果如下：

图 1-21

```
[root@kotlin ~]# ls
anaconda-ks.cfg  jdk-8u131-linux-x64.tar.gz
```

（3）创建 Java 目录，把刚刚上传的 JDK 压缩包解压缩到 Java 目录下。解压缩完成后，在 Java 目录下会多出一个 jdk1.8.0_131 目录。命令如下：

```
[root@kotlin ~]# mkdir java
[root@kotlin ~]# tar -xf jdk-8u131-linux-x64.tar.gz -C java
[root@kotlin ~]# cd java/
[root@kotlin java]# ls
jdk1.8.0_131
```

（4）接下来配置 JDK 环境变量。使用 vim 编辑器打开/etc/profile 文件，将光

标移动到最后一行，按 O 键进行编辑，添加如下代码：

```
[root@kotlin java]# vim /etc/profile
JAVA_HOME="/root/java/jdk1.8.0_131"
CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$PATH
```

提示：JAVA_HOME 环境变量的值是第 4 步解压缩之后生成的 jdk1.8.0_131 目录。

编辑后需要重启服务器才能生效，但是重启会很麻烦，使用 source 命令可以让/etc/profile 文件直接生效。命令如下：

```
[root@kotlin java]# source !$
source /etc/profile
```

（5）使用“java-version”命令查看 JDK 是否安装成功，结果如下：

```
[root@kotlin java]# java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

1.2.2 在 Linux 操作系统下安装及配置 Kotlin

（1）下载 Kotlin 安装包。在 Linux 操作系统下下载的 Kotlin 安装包和在 Windows 操作系统下所下载的是一样的，所以使用 rz 命令将之前在 Windows 操作系统下下载的 Kotlin 安装包上传到 Linux 操作系统中即可。

（2）解压缩 kotlin-compiler-1.1.2-5.zip，在 Java 目录下会多出一个 kotlinc 目录。命令如下：

```
[root@kotlin java]# unzip kotlin-compiler-1.1.2-5.zip
[root@kotlin java]# ls
jdk1.8.0_131  kotlinc  kotlin-compiler-1.1.2-5.zip
```

（3）接下来配置 Kotlin 环境变量。使用 vim 编辑器打开/etc/profile 文件，将光标移动到最后一行，按 O 键添加如下代码，然后使用 source 命令使/etc/profile 文件生效。

```
[root@kotlin java]# vim /etc/profile
KOTLIN_HOME="/root/java/kotlinc"
PATH=$KOTLIN_HOME/bin:$PATH
```

```
[root@kotlin java]# source /etc/profile
```

提示：KOTLIN_HOME 环境变量的值是第 2 步解压缩之后生成的 kotlinc 目录。

（4）上述操作完成后，在命令行输入“kotlin –version”，若显示版本号则表示 Kotlin 安装成功。命令如下：

```
[root@kotlin java]# kotlin -version
Kotlin version 1.1.2-5 (JRE 1.8.0_131-b11)
```

1.3　编写第一个 Kotlin 程序：Hello Kotlin

无论学习哪一种编程语言，在第一个 Hello Kotlin 程序运行成功后，都代表迈入学习该语言的第一步。在学习 Kotlin 之前，读者一定要在计算机上完成 Kotlin 环境的配置；否则，接下来的内容将无法完成。

（1）新建一个记事本，并命名为以.kt 为扩展名的文件，如 HelloKotlin.kt。添加如下代码：

```
fun main(args:Array<String>)
{
    println("Hello Kotlin")
}
```

代码释义如下。

- fun：用于定义一个函数。
- main：函数名称。
- args：函数中的参数，这个参数的数据类型是数组（array）。
- println：打印字符串并换行。

（2）在 Windows 系统下的 DOS 命令行中输入如图 1-22 所示的命令，将在同级目录下生成一个 HelloKotlin.jar 文件，如图 1-23 所示。

图 1-22

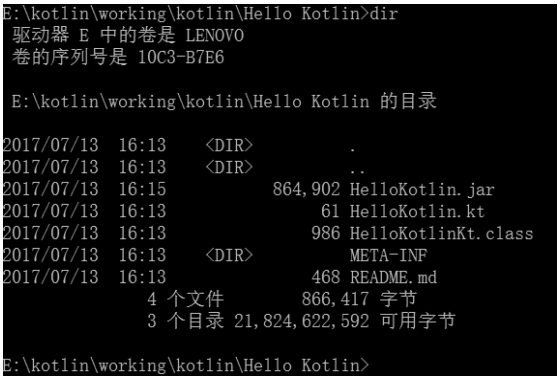


图 1-23

命令详解：

- -d 选项，用于指定所生成的 HelloKotlin.jar 文件，该 JAR 包可以通过 java -jar 命令运行。
- -include-runtime 选项，指定在所生成的 JAR 包中包含 Kotlin 包，这样这个 JAR 包就可以独立运行了。如果不添加该选项，那么所生成的 JAR 包就不能独立运行，但是可以将 JAR 包导入到含有 Kotlin 包的项目中作为库使用。

1.4 Kotlin 程序运行过程

Kotlin 是基于 Java 虚拟机（JVM）运行的，Kotlin 编译器生成的 JVM 字节码与 Java 编译的字节码基本相同，因此，Kotlin 与 Java 可以完全兼容，并且 Kotlin 的语法更加简洁。Kotlin 和 Java 有如下三个相同点：

- 源文件（HelloKotlin.kt）经过 Kotlin 编译后生成.class 文件，如 HelloKotlinKt.class。
- 源文件经过打包之后生成.jar 文件，如 HelloKotlin.jar。
- 在生成 HelloKotlin.jar 文件之前，如果包含了 Kotlin 包，那么该 JAR 包可以独立运行；反之，则不能独立运行，但是可以将该 JAR 包导入到含有 Kotlin 包的项目中作为库使用。

1.5　使用 IDE 编译并运行 Hello Kotlin 程序

因为 Kotlin 和 ideaIC 同出自 JetBrains 公司，所以需要下载官方的 IDE 编辑器，使用 IDE 编辑器创建 Hello Kotlin 项目。整个流程如下：

（1）下载并安装 IDE 编辑器。

（2）使用 JetBrains ideaIC 编辑器创建 Hello Kotlin 项目。

（3）安装 Kotlin 插件。

如果使用的是 JetBrains ideaIC 编辑器，则可能不需要安装 Kotlin 插件；但如果使用的是其他 IDE 编辑器，如 Android Studio、Eclipse 等，则需要安装 Kotlin 插件。

插件下载地址：https://plugins.jetbrains.com/plugin/6954-kotlin。

1.5.1　安装 JetBrains ideaIC 编辑器

安装 JetBrains ideaIC 编辑器的具体操作步骤如下。

（1）进入 JetBrains 官方网站（http://www.jetbrains.com/idea/download/#section=window），下载编辑器，如图 1-24 所示。其中 UItimate 版本的功能强大，是收费的；Community 版本的功能比较少，是免费的。

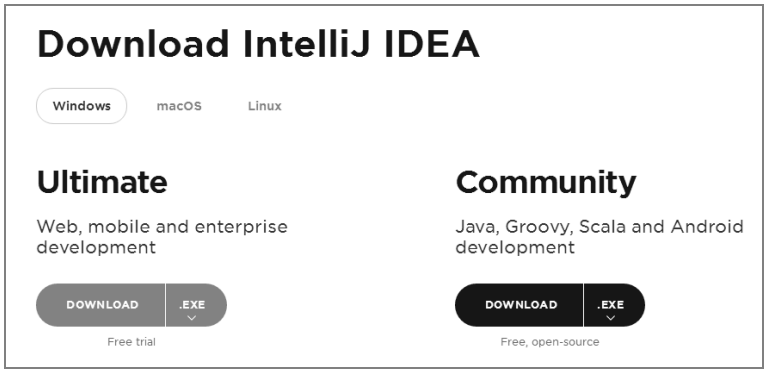


图 1-24

（2）双击下载文件 ideaIC-2017.1.4.exe 进行安装，单击“Next”按钮，进入如图 1-25 所示的窗口，根据自己的需要选择安装目录。

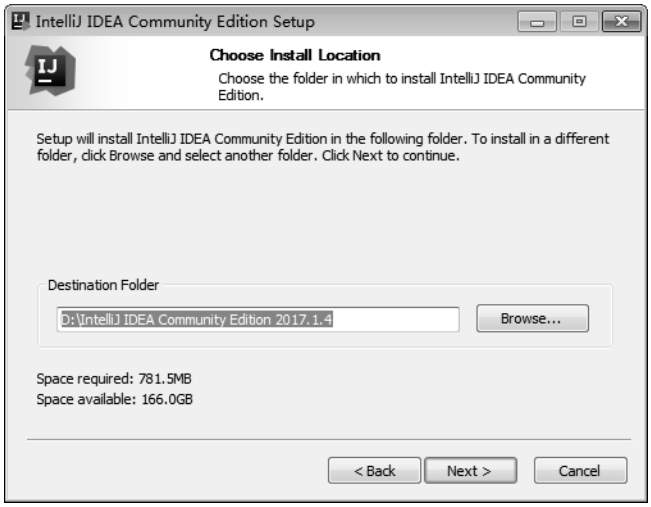


图 1-25

（3）单击“Next”按钮，进入 IDE 配置选项窗口，设置是否创建桌面快捷方式和关联文件扩展名，这里勾选全部选项，如图 1-26 所示。

图 1-26

（4）单击“Next”按钮，进入选择开始菜单文件夹窗口，保持默认设置即可，最后单击“Install”按钮进行安装。安装完成后会弹出如图 1-27 所示的窗口，该窗口中有一个“Run IntelliJ IDEA”选项，如果勾选该选项后单击“Finish”按钮，则会在安装后运行该 IDE 编辑器；如果不勾选该选项，则会关闭安装窗口，表示

ideaIC-2017.1.4.exe 程序已经安装完成。

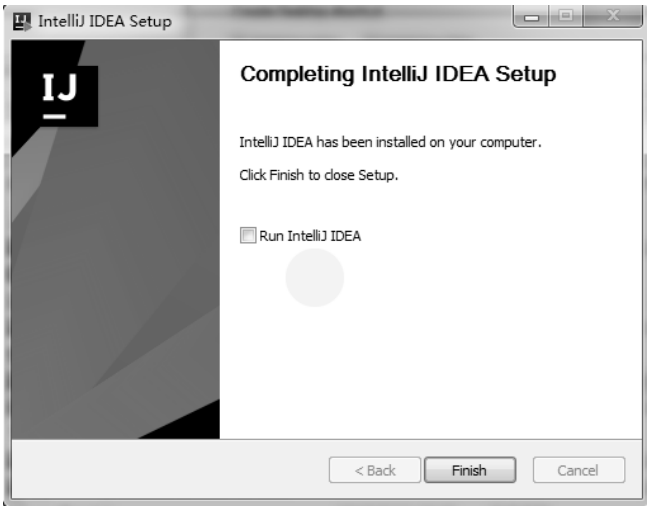


图 1-27

1.5.2　在 JetBrains ideaIC 编辑器中安装 Kotlin 插件

在 JetBrains ideaIC 编辑器中安装 Kotlin 插件的具体步骤如下。

（1）JetBrains ideaIC 编辑器安装成功后，会在桌面上生成一个快捷方式，如图 1-28 所示。

图 1-28

（2）双击执行该程序，将出现如图 1-29 所示的窗口，选择默认选项，单击“OK”按钮后，弹出如图 1-30 所示的窗口，输入从官方购买的注册码进行正版授权。

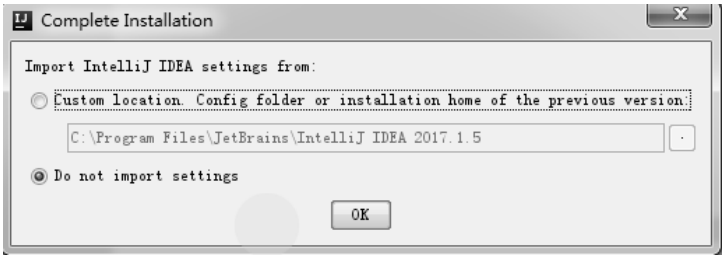


图 1-29

（3）注册完成后，出现如图 1-31 所示的欢迎界面。

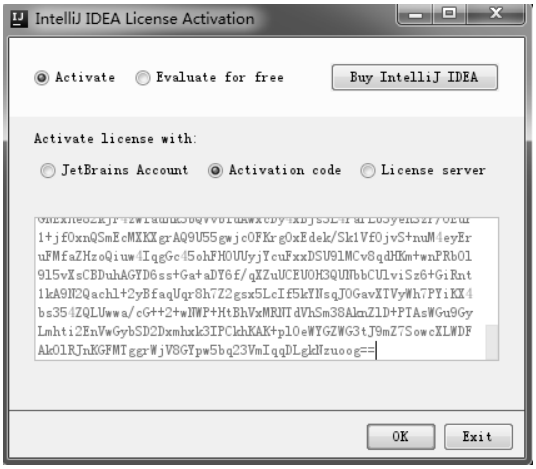


图 1-30

图 1-31

（4）单击右下角的“Configure”，在下拉列表中选择“Plugins”选项，打开插件列表，如图 1-32 所示。

（5）单击“Install JetBrains plugin…”按钮，安装 JetBrains 插件列表，在搜索框中输入“Kotlin”关键词，如图 1-33 所示。

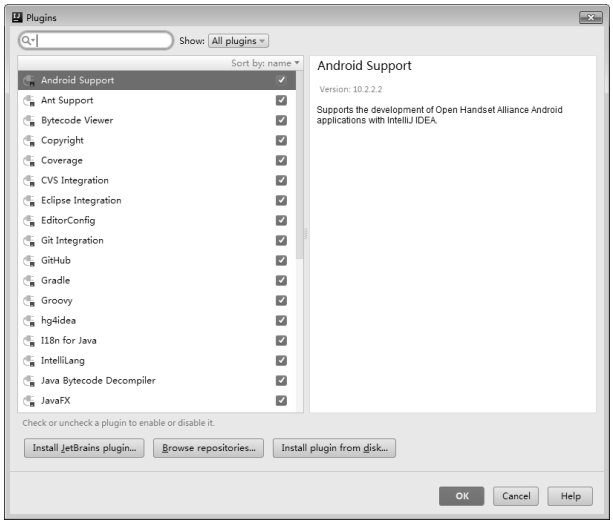


图 1-32

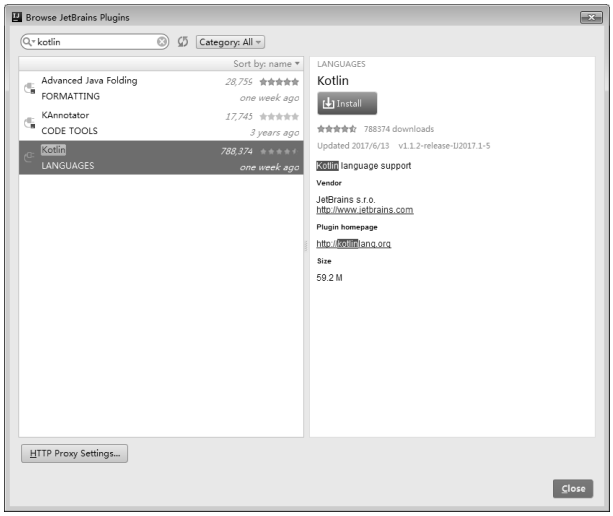


图 1-33

（6）单击右边的“Install”按钮，或者双击左边列表中的插件进行安装，安装过程非常缓慢，需要耐心等待。安装完成后返回搜索“kotlin”关键词的页面，如果右边已经没有“Install”按钮了，则表示安装成功，重启 IDE 编辑器就可使用了。

1.5.3 在 JetBrains ideaIC 中创建“Hello Kotlin”项目

在 JetBrains ideaIC 的欢迎界面中有如下选项：Create New Project（创建新项目）、Import Project（导入项目）、Open（打开项目，主要针对单个文件）、Check out from Version Control（查看版本控制，其列表中有 Git、CVS、Github、Mercurial、Subversion 选项）。在 JetBrain ideaIC 中创建“Hello Kotlin”项目的具体步骤如下。

（1）在欢迎界面中单击“Create New Project”选项，打开如图 1-34 所示的新建项目对话框，单击左侧的“Kotlin”，选择右侧列表中的 Kotlin(JVM)即可。

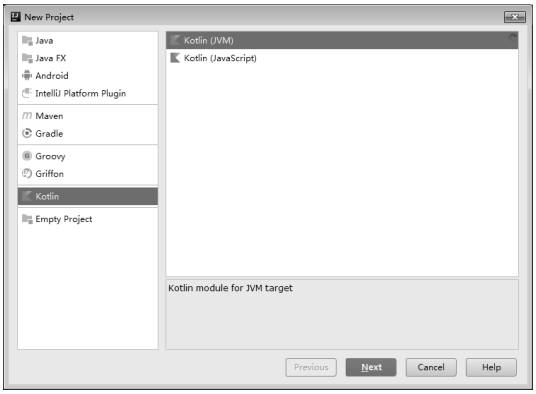


图 1-34

（2）单击“Next”按钮，进入项目基本设置界面，如图 1-35 所示，其中“Project name”表示项目名称，“Project location”表示项目存放目录，“Project SDK”表示项目 SDK 的安装目录。

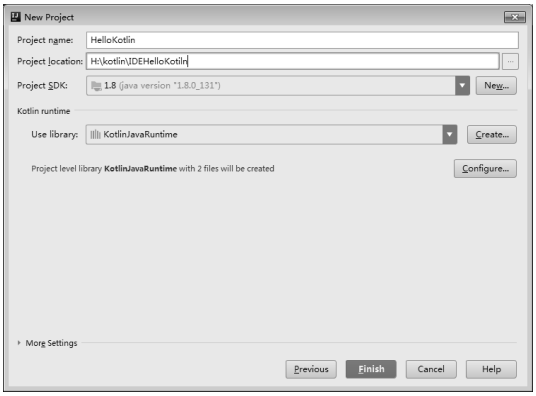


图 1-35

（3）填写完毕后，单击“Finish”按钮，打开如图 1-36 所示的界面，发现在 Project 列表中已经有了目录和文件。有 Java 基础的读者，一般都知道 src 目录是存放代码的地方；没有 Java 基础的朋友，现在按照步骤进行操作也会知道 src 目录是存放代码的地方。

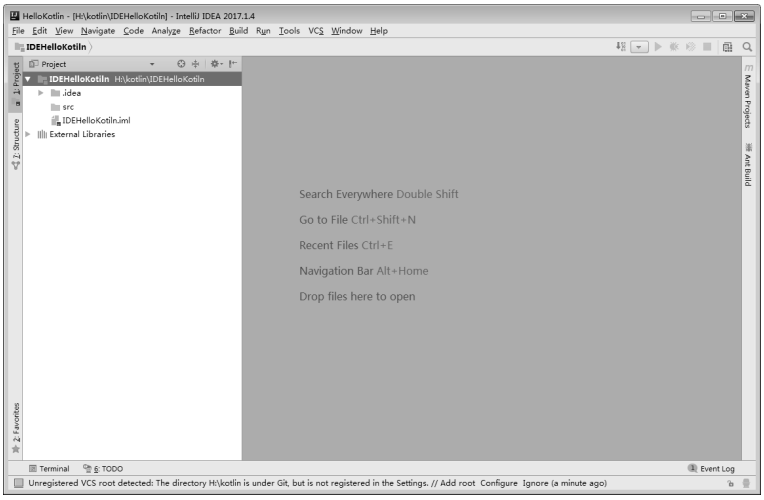


图 1-36

（4）选择 src 目录并单击鼠标右键，在弹出的快捷菜单中选择新建包（Package）选项，在打开的窗口中输入包名“jqiang.helloKotlin”，然后单击“OK”按钮完成创建，如图 1-37 所示。

（5）选择 jqiang.helloKotlin 包并单击鼠标右键，在弹出的快捷菜单中选择新建 Kotlin 文件选项，在打开的窗口中输入 Kotlin 文件名为“HelloKotlin”，如图 1-38 所示。

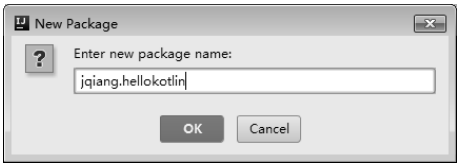


图 1-37

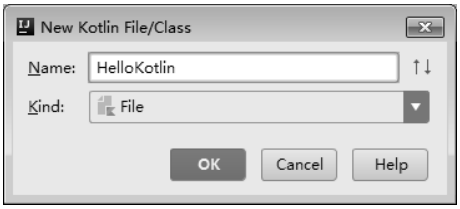


图 1-38

（6）创建完成之后，可以在主函数 main 中使用 println 换行打印出“Hello Kotlin”，如图 1-39 所示。

图 1-39

（7）单击菜单“Run”→“Run 'jqiang.hellokotlin.HelloKotlin…'”，或按“Alt+Shift+F10”快捷键（在不同的操作系统下快捷方式不同，请确认后再使用），即可开始执行该项目，如图 1-40 所示。

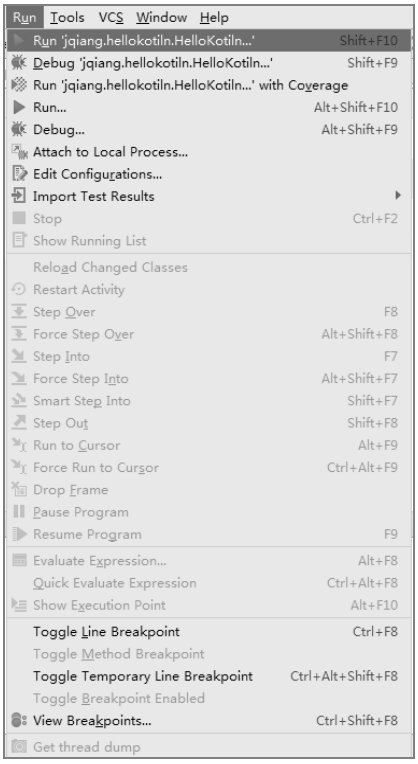


图 1-40

（8）执行完毕后，在命令控制台换行打印出“Hello Kotlin”，如图 1-41 所示，表示“Hello Kotlin”项目已经成功完成。

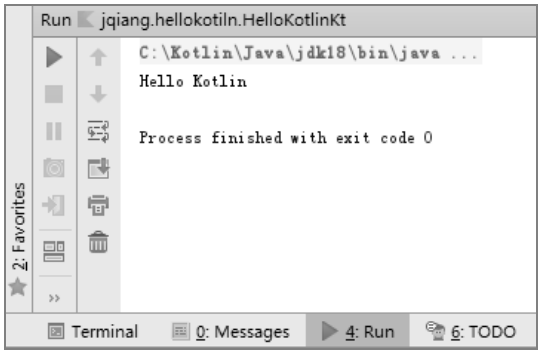


图 1-41

1.6　本章小结

本章主要介绍了 JDK 和 Kotlin 在不同操作系统下的安装及配置过程，并通过不同方式运行了“Hello Kotlin”项目。但是现在就使用命令行去编译较大的项目并不现实，所以接下来将详细讲解 Kotlin 语言的具体应用。

第2章 Kotlin 编程基础

学习任何语言都要先了解其基础知识，学习 Kotlin 语言也是一样的，只有把基础知识学会并加以巩固，才能进阶、学习更高级的编程。

本章主要内容：

- Kotlin 编程风格
- Kotlin 变量和常量
- Kotlin 数据类型
- Kotlin 运算符
- Kotlin 流程控制语句

2.1 Kotlin 编程风格

每一种编程语言都有其基本格式，Kotlin 也不例外。虽然 Kotlin 并没有像 Java 那样有着严格的编码规范，但它也有自己的规范和风格。

（1）Kotlin 的命名规则与冒号的使用

- 驼峰命名法（不要使用下画线），如：public class DataBaseUser。
- 类的首字母要大写，如：class People。
- 方法和属性名的首字母要小写，如 fun say(){}。
- 代码缩进统一为 4 个字符。
- 冒号是子类继承父类使用的符号，在使用时前后需要加空格，而在类实例化时无须使用空格隔开。例如：

```
interface Foo<out T : Any> : Bar {
```

```
        fun foo(a: Int): T
    }
```

（2）Lambda 表达式

- 在 Lambda 表达式中，在花括号后要使用空格隔开，而且箭头前后也要有空格，以便于分开参数与函数体，如：

```
    list.filter { it > 10 }.map { element -> element * 2 }
```

（3）Kotlin 语法风格

- 变量定义：

```
    val|var<propertyName>[: <PropertyType>] [= <property_initializer>]
[<getter>][<setter>]
```

- 函数定义：

```
    fun sum(a: Int, b: Int): Int { return a + b }
```

- 字符串模板：

```
    val a = "this is String" println(${a})
```

- Kotlin 的注释方式，与 Java、JavaScript 等编程语言的注释方式一样：

```
    // 单行注释
    /**
    * 多行注释
     */
```

2.2　Kotlin 常量和变量

在 Kotlin 中，常量必须用 val 关键字来声明，而变量必须用 var 关键字来声明，它们的声明方式如下：

```
    val|var <propertyName>[: <PropertyType>] [= <property_initializer>]
[<getter>][<setter>]
```

在上述声明中，val 和 var 是必选参数，而且是二选一的；propertyName 是必选参数，在声明变量名称时使用；PropertyType 是可选参数，表示声明变量的数据类型；property_initializer 是必选参数，表示声明变量的初始值；getter 和 setter 是可选参数，表示设置和获取。

2.2.1 常量

常量的值是不可改变的，即常量被定义后，在程序的任何地方都不能改变它的值。在其他编程语言中，对常量名有一定的限制（比如在 PHP 中常量名是由英文字母、下画线和数字组成的），但是在 Kotlin 中常量名不再局限于英文字母、下画线和数字，还可以是中文文字。虽然对 Kotlin 的常量名没有过多的限制，但是我们在给常量命名时要做到见名知意。

【例 2.1】常量的定义和使用。

```
package jqiang.amount
    val constant1=5.5
    //val constant2: Int?
    val constant3: Int=10
    val 常量 =10
fun main(args: Array<String>){
    println("constant1 的值是"+constant1)
    // println("constant2 的值是"+constant2)
    println("constant3 的值是"+constant3)
    println(常量)
}
```

运行结果如下：

```
constant1 的值是 5.5
constant3 的值是 10
10
```

2.2.2 变量

变量的值可以被改变，它没有固定值，即变量被定义后，在程序的任何地方都可以改变它的值。变量的值离程序结果最近，即为该变量的最终值。

【例 2.2】变量的定义和使用。

```
package jqiang.amount
    var var1: String="Kotlin"
    //var var2: Int?
    var var3: Int=10
    var var4=5
    set(value) {
       if (value <= 5)
       field = value
```

```
    }
fun main(args: Array<String>){
    println("var1 不重新赋值情况是"+var1)
    println("var1 的值是"+var1)
    var1=var1+5.3
    println("var1 的值是"+var1)
    // println("var3 的值是"+var)
    var3=var3+5
    println("var3 的值是"+var3)
    var4=var4+5
    println("var4 的值是"+var4)
}
```

运行结果如下：

```
var1 不重新赋值情况是 Kotlin
var1 的值是 Kotlin5.3
var3 的值是 15
var4 的值是 5
```

在上述示例中，细心的读者会发现定义常量和定义变量的方式是一样的，并且在定义 var4 时加了一个条件限制——当其值小于或等于 5 时，备用字段使用 field 关键字声明，field 关键字只能用于属性的访问。这里的“+”不仅可以作为字符串连接符，而且可以作为加法运算符。

2.2.3　变量作用域

变量根据作用域分为全局变量和局部变量，其中全局变量是指类中的成员变量；局部变量是指在方法中定义的变量。无论是全局变量还是局部变量都需要进行初始化。

【例 2.3】变量的作用域。

```
package jqiang.amount
class vals(val a: Int =5){
    fun geta(): Int = a
}
fun varfun(): Int {
    var b=10
    return b
}
var c=15
fun main(args: Array<String>) {
```

```
    // println(a) // Unresolved reference: a
       println(vals().a)
    // println(b) // Unresolved reference: a
       println(varfun())
       println(c)
    }
```

运行结果如下：

```
5
10
15
```

在一个类中，在析构函数中定义一个变量，如果将该变量放在外部，则需要在该类中定义一个方法返回该变量的值；如果在类中直接定义变量，则可以先实例化对象，然后直接访问该成员变量。

定义在函数中的变量属于局部变量，如果在外部调用该变量的值，则需要在该方法中使用 return 函数返回该变量的值。一个函数的作用就是指一个小功能通过函数内部进行业务处理之后将结果返回，如果直接将刚定义的变量返回，则实在有些浪费。

2.3 Kotlin 数据类型

Kotlin 的最基本元素是数字和字符，要学会使用数字和字符编写程序，则必须掌握其语法规则。本节重点介绍 Kotlin 编程语言中的数据类型。

在 Kotlin 编程语言中有两种数据类型：基本数据类型和引用数据类型。

- 基本数据类型包括：布尔类型（Boolean）、数值类型（Number）、字符类型（Char）。
- 引用数据类型包括：可空类型（Null）、对象类型（Object）和数组类型（Array）。

2.3.1 布尔类型（Boolean）

布尔类型是比较常见的数据类型，它只有 true 和 false 两个值，设置为布尔类型只需将 true 和 false 赋值给变量即可。

在 Kotlin 中，布尔类型使用 Boolean 表示，如 val boo：Boolean=true。

【例 2.4】布尔类型变量通常被应用在条件控制或者循环控制语句的表达式中。在下面的 if 条件控制语句中判断变量 boo 是否为 true，如果为 true，则输出“变量 boo 为真”；否则输出“变量 boo 为假”。

```
package jqiang.Boolean
var boo: Boolean=true
fun main(args: Array<String>){
    if(boo==true){
        println("变量 boo 为真！")
    }else{
        println("变量 boo 为假！")
    }
}
```

运行结果如下：

```
变量 boo 为真！
```

在 Kotlin 中提供了内置的布尔类型运算，如逻辑或（||）、逻辑与（&&）及逻辑非（!）。

2.3.2　数值类型（Number）

Number 类型包括整型和浮点型，可以将整型理解为整数，将浮点型理解为小数。

- 整型：包含最小的 16 位的 short 类型，最常见的 32 位 int 类型和最大 64 位长整型 long 类型。
- 浮点型：包含 64 位双精度浮点型 double 类型和 32 位的 float 类型。

字节类型其实也属于数值类型，8 位，只是在程序中很少会用到这种数据类型，它一般用于数据流的数据记载。

Kotlin 的数值类型如表 2-1 所示。

表 2-1

数值类型		占用字节	取值范围
整型	短整型（Short）	1	[-32768,32767]
	整型（Int）	4	[-2147483649,2147483647]
	长整型（Long）	8	[-9223372036854775808,9223372036854775807]
浮点型	双精度浮点型（Double）	8	[4.9E-324,1.7976931348623157E308]
	单精度浮点型（Float）	4	[1.4E-45,3.4028235E38]
字节类型（Byte）		1	[-128,127]

【例 2.5】常见的 Number 类型值。

```
    package jqiang.Number
    var int: Int=8//声明十进制形式的整数
    var teint: Int=0x88888//声明十六进制形式的整数
    var maxint: Int= Int.MAX_VALUE//声明变量 maxint，将 Int 类型的最大值赋值
给 maxint
    var minint: Int= Int.MIN_VALUE//声明变量 minint，将 Int 类型的最小值赋值
给 minint
    var long: Long=128L//声明十进制形式的长整型变量
    var maxlong: Long= Long.MAX_VALUE//声明变量 maxlong，将 Long 类型的最大
值赋值给 maxlong
    var minlong: Long= Long.MIN_VALUE//声明变量 minlong，将 Long 类型的最小
值赋值给 minlong

    var double: Double= 2.88//声明双精度符点类型变量
    var maxdouble: Double= Double.MAX_VALUE//声明变量 maxdouble，将 Double
类型的最大值赋值给 maxdouble
    var mindouble: Double= Double.MIN_VALUE//声明变量 mindouble，将 Double
类型的最小值赋值给 mindouble

    var float: Float=2.0f//声明 Float 类型
    var maxfloat: Float= Float.MAX_VALUE//声明变量 maxfloat，将 Float 类型
的最大值赋值给 maxfloat
        var minfloat: Float= Float.MIN_VALUE//声明变量 minfloat，将 Float
类型的最小值赋值给 minfloat

    var maxshort: Short= Short.MAX_VALUE//声明变量 maxshort，将 Short 类型
的最大值赋值给 maxshort
    var minshort: Short= Short.MIN_VALUE//声明变量 minshort，将 Short 类型
的最小值赋值给 minshort

    var maxbyte: Byte= Byte.MAX_VALUE//声明变量 maxbyte，将 Byte 类型的最大
值赋值给 maxbyte
    var minbyte: Byte= Byte.MIN_VALUE//声明变量 minbyte，将 Byte 类型的最小
值赋值给 minbyte

    fun main(args: Array<String>){
        println("int 十进制数值: "+int)
        println("int 十六进制数值: "+teint)
        println("int 最大值: "+maxint)
        println("int 最小值: "+minint)
```

```
        println("long: "+long)
        println("long 最大值: "+maxlong)
        println("long 最小值: "+minlong)

        println("double: "+double)
        println("double 最大值: "+maxdouble)
        println("double 最小值: "+mindouble)

        println("float: "+float)
        println("float 最大值: "+maxfloat)
        println("float 最小值: "+minfloat)

        println("short 最大值: "+maxshort)
        println("short 最小值: "+minshort)

        println("byte 最大值: "+maxbyte)
        println("byte 最小值: "+minbyte)
    }
```

运行结果如下：

```
int 十进制数值：8
int 十六进制数值：559240
int 最大值：2147483647
int 最小值：-2147483648
long：128
long 最大值：9223372036854775807
long 最小值：-9223372036854775808
double：2.88
double 最大值：1.7976931348623157E308
double 最小值：4.9E-324
float：2.0
float 最大值：3.4028235E38
float 最小值：1.4E-45
short 最大值：32767
short 最小值：-32768
byte 最大值：127
byte 最小值：-128
```

在上面例子中有几个特殊的值，如 0x88888、2.0f，这样的值在 Kotlin 中有一个具体的名称，即字面常量。

在 Kotlin 中常见的字面常量有：

- 数字常量和浮点常量，如十进制长整型（123L）、十六进制（0x0f）、二

进制（0b00001011）等形式的常量。

- 浮点常量，如双精度浮点类型（12.5e10）、浮点类型（12.5f）等常量。
- 下画线数字类型，如 1_000_000、123_456L、0xFF_EC_DE_5E 等含有下画线的数字。

2.3.3 字符类型（Char）

字符常量是由一对单引号围起来的单个字符，可以是 16 位的 Unicode 字符集中的任意字符，如：'a'、'0'、'$'等，也可以是转义字符（\），或者直接写出字符编码。

在 Kotlin 中，字符类型用 Char 表示，如 val char：Char='a'。

【例 2.6】字符类型的使用。

```
package jqiang.Char
fun main(args: Array<String>) {
    val letterchar: Char='a'
    val nubchar: Char='0'
    val Unicodechar: Char='\u7231'
    val yi: Char='\n'
    println("转义字符${yi}字母,char 类型${letterchar},数字 char 类型${nubchar},Unicode 编码 char 类型：${Unicodechar}")

}
```

运行结果如下：

```
转义字符
字母,char 类型 a,数字 char 类型 0,Unicode 编码 char 类型：爱
```

在上面例子中，为常量 yi 赋的值是“\n”，但输出的是换行操作，这就是转义字符的作用。

转义字符是具有特殊含义和功能的字符，Kotlin 中的所有转义字符都以反斜杠（\）开头，后面跟着具有特定含义的字符。表 2-2 中列出的是常见的转义字符。

表 2-2

转义字符	含　义
\t	制表符
\r	回车
\n	换行

续表

转义字符	含　义
\’	单引号
\”	双引号
\\	反斜杠
\$	美元符号，在 Kotlin 中支持美元符号的模板

2.3.4　字符串类型（String）

字符串是连续的字符序列，由数字、字母和字符组成。在 Kotlin 中字符串有两种字面值，转义字符串可以是转义字符，原生字符串可以包含换行和任意文本，转义字符同样需要用反斜线的方式来注明，原生字符串需要使用三个引号分界符括起来。

在 Kotlin 中，字符串使用 String 表示，字符串是不可变的，并且必须使用双引号围起来，如 val char：String="this is a String"。

【例 2.7】定义字符串。

```
   package jqiang.String
   fun main(args: Array<String>) {
       val string: String="Hello \'Kotlin\'"
       val chararr: String=String(charArrayOf('H', 'e', 'l', 'l', '\t',
'K', 'o', 't', 'l', 'i', 'n'))
       println("string 字符串"+string+"字符串拼接"+chararr)
       val a: Int=1
       val b: Int=0
       println("Java 计算方式: "+a+"+"+b+"="+(a+b))
       println("Kotlin 计算方式: $a+$b=${a+b}")
       val money: Int=888
       println("方式一: $$money "+"方式二: "+"$"+money)
  }
```

运行结果如下：

```
   string 字符串 Hello 'Kotlin'字符串拼接 Hell Kotlin
   Java 计算方式: 1+0=1 Kotlin 计算方式: 1+0=1
   方式一: $888 方式二: $888
```

字符串可以包含模板表达式，并且可以求值并把结果合并到字符串中。模板表达式一般以“$”符号开头，后面跟变量名，如$i；或者使用花括号括起来，如

${a+b}。

原生字符串和转义字符串内部都支持字符串模板，如果要在原生字符串中表示“$”符号，则可以使用${'$'}money。

字符串连接使用“+”符号，如$$money"+"$+money。

2.3.5 元组类型（Tuple）

元组是一种简单的类型，它可以把不同类型的值组合在一起，组成元组的数据就被称为“元素”。在 Kotlin 中，元组分为二元元组（Pair）和三元元组（Triple）。

【例 2.8】（404，"Not Found"）是一个描述 HTTP 状态码（HTTP Status Code）的元组。HTTP 状态码是请求网页时 Web 服务器返回的一个特殊值，如果请求的网页不存在，则会返回状态码 404。

```
   packagejqiang.Tuple
    fun main(args: Array<String>) {
        //访问 Tomcat+PHP+MySQL 环境，反馈得到状态码 404，表示不存在该文件
        val (status, msg)=Pair(404, "Not Found")
        val (server, script, database)=Triple("Tomcat", "PHP",
"MySQL")
        println("方式一：访问${server}+${script}+${database}环境，
反馈得到状态码${status}，表示${msg}")

         val http=Pair(404, "Not Found")
         val Development=Triple("Tomcat", "PHP", "MySQL")
         println("方式二：访问${Development.first}+${Development.
second}+${Development.third}环境，反馈得到状态码${http.first}，表
示${http.second}")
    }
```

运行结果如下：

```
方式一：访问 Tomcat+PHP+MySQL 环境，反馈得到状态码 200，表示 Not Found
方式二：访问 Tomcat+PHP+MySQL 环境，反馈得到状态码 200，表示 Not Found
```

2.3.6 可空类型（Null）

在编程开发过程中，经常会碰到变量为空值的情况，如果该变量是引用类型的，那么它的设置就是 Null；如果是值类型的，那么在不使用可空类型的情况下，

会像 Java 一样抛出空值的异常。为了避免发生这种情况，Kotlin 可以创建一个可空类型的变量。

【例 2.9】定义一个可空类型的变量。

```
package jqiang.Null
fun main(args: Array<String>) {
    var address: String?= null//可空类型，可以赋值为 null
      println(address?.length)
}
```

在项目中，经常会从数据库中查询出来可空类型数据，因此在 Kotlin 中提供了以下方式来访问可空类型数据。

- 条件判断：如 var b=if(address!=null)address.length else -1，通过条件语句判断该变量是否为空值，如果不为空值，则返回该变量；否则，就返回该变量的长度或值。
- 使用安全操作符（?.）：如 address?.length，如果 address 为非空值，则返回 a 的长度或其他值；否则，就返回 null。
- 使用 Elvis 操作符（?:）：如 address?.length?:-1，如果 address 为非空值，则使用变量 address；否则，就使用某个非空值。
- 使用“!!”操作符：如果想像 Java 一样抛出空值异常，就必须对变量显式使用“!!”操作符，如 a!!。
- 类型转换：如果对象的类型不是目标类型，那么进行常规类型转换会导致 ClassCastException 异常，所以需要进行安全类型转换，如 var b:Int?=a as? Int。

当集合中出现空值时，可以使用 FilterNotNull 函数去除集合中为空的元素，重新生成没有空值的集合。

2.3.7　对象类型（Object）

对象类型是存储数据和有关如何处理数据的数据类型。在 Kotlin 中必须明确声明对象，首先声明对象的类，类是包含属性和方法的结构。声明对象使用 class 关键字。使用该对象时，首先要进行实例化，然后才能使用对象中的成员属性和方法。

【例 2.10】对象类型的使用。

```
package jqiang.Object
```

```
class course (var name: String){
    init {
        println("您选择的课程名称是${name}")
    }
}
fun main(args: Array<String>) {
    val studentobj=course("Kotlin")
}
```

运行结果如下：

```
您选择的课程名称是 Kotlin
```

2.3.8 数组类型（Array）

数组是具有相同类型的变量的集合，这些变量具有相同的标识符，即数组名；数组中的每一个变量都称为元素。要引用数组中的特定元素，通常使用数组名加上一个用中括号（[]）括起来的整型表达式来表示，该表达式被称为数组的索引（index）或下标，如 ArrayA[8]，其中 ArrayA 是数组名，数字 8 是数组的索引。数组中第一个元素的索引值是 0，第二个元素的索引值是 1，依此类推，如 ArrayA[8] 表示 ArrayA 数组中的第 9 个元素。

在 Kotlin 中，数组位于 Array 类中，由 get 和 set 函数、size 属性以及其他成员函数表示，代码如下：

```
package Kotlin
public class Array<T> {
    public inline constructor(size: Int, init: (Int) -> T)
    public operator fun get(index: Int): T
    public operator fun set(index: Int, value: T): Unit
    public val size: Int
    public operator fun iterator(): Iterator<T>
}
```

可以使用 arrayof()创建一个数组，将值传递给它；也可以使用 arrayOfNulls() 库函数创建一个长数组。

注意：Kotlin 中的数组是不变的，这意味着 Kotlin 没有给 Array<String>分配到 Array<Any>，但是 Kotlin 为了避免这种情况的发生，可以直接使用 Array<out Any>。

【例 2.11】由三种不同数据类型元素组成的数组。

```
package jqiang.Array
    val arrayOfInt: IntArray = intArrayOf(1, 2, 3, 4, 5)
    val arrayOfChar: CharArray = charArrayOf('H', 'e', 'l', 'l',
'o')
val arrayOfString: Array<String> = arrayOf("我", "爱", "Kotlin")
fun main(args: Array<String>) {
    println(arrayOfInt[2])
    for(String in arrayOfString){
        println(String)
        }
    }
```

输出结果如下：

```
3
我
爱
Kotlin
```

在上面例子中，“[]”表示调用成员函数 get()和 set()，即“[]”是通过索引访问元素的。

声明数组后，数组中的元素个数可以自由更改，只要对该属性数组进行赋值，数组的长度就会自动增长。还可以对数组进行常规操作。

2.3.9　数据类型的检查和转换

无论是强类型的编程语言（如 Java），还是弱类型的编程语言（如 PHP），都会进行类型转换，Kotlin 也不例外，它提供了 5 种数据类型转换方式，即智能转换、不安全转换、安全转换、显式转换和隐式转换。

【例 2.12】数据类型的检查和转换。

```
package jqiang.datatype
fun main(args: Array<String>) {
    val a=10
    if (a is Int ) {
        println(a)
    }
    if (a !is Int ) {
        println(a)
    }
```

```
    val b: String?=a as? String
    println(b)
    val c=a.toDouble()
    println(c)

}
```

运行结果如下：

```
10
null （大小写）
10.0
```

智能转换：在 Kotlin 中不需要使用转换操作符进行操作，因为编辑器会跟踪不可变值的 is 来进行检查，并且在需要时会自动插入进行安全转换。比如 var a:Int=10，如果将 Int 类型更改为 String 类型，此时在编辑器中在 String 下面会出现红色波浪线，将鼠标指针放上去，则会提示“Incompatible Types: String and Int”，表示该变量定义出现错误。解决方法是，更改等号左边的值，用双引号引起来，表示该值为字符串类型，或者将 String 类型改成 Int 类型。虽然编辑器可以自动识别数据类型，但还是有一定局限性的——无法识别被 open 关键字修饰的属性，或者自定义 getter 属性的 val 值；无法识别被修改的 var 局部变量；var 变量永远无法识别（因为该变量随时在被其他代码更改）。

不安全转换：通常可以进行转换，但是如果碰到不可能被转换的情况或者为 Null 值时，则会抛出异常，不建议转换。如 val a: String = b as String。

安全转换：为了避免不安全转换抛出异常，可以使用转换操作符 as?。如 val b: String?=a as? String，当转换失败时会返回 Null，虽然右边是一个非空的字符串，但是返回的也是空值。

显式转换：因为不能将较小的数据类型隐式转换为较大的类型，这就意味着程序无法继续执行，所以每个数字都支持进行类型转换。在 Kotlin 中提供的数字转换函数如表 2-3 所示。

表 2-3

函　数	示　例	说　明
toByte()	var a=10 a.toByte()	转换字节
toShort()	var a=10 a.toShort()	转换短整型
toInt()	var a=10 a.toInt()	转换整型
toLong()	var a=10 a.toLong()	转换长整型

续表

函　数	示　例	说　明
toFloat()	var a=10 a.toFloat()	转换浮点型
toDouble()	var a=10 a.toDouble()	转换双精度浮点型
toChar()	var a=10 a.toChar()	转换字符型

隐式转换：在运算的过程中进行转换，如 val a=1L+3。

2.4　Kotlin 运算符

运算符是用来对变量、常量及数据进行计算的符号，可以使用运算符对一个值或者一组值执行指定的操作。Kotlin 中的运算符包括算术运算符、关系运算符、逻辑运算符、赋值运算符等。Kotlin 中还提供了位运算函数来进行位运算。

2.4.1　算术运算符

算术运算符通常使用在数字处理中，对数值类型的数据进行运算。Kotlin 中的算术运算符分为一元运算符和多元运算符，如表 2-4 所示。

表 2-4

算术运算符	说　明	示　例
+	加法	a + b
-	减法	a - b
*	乘法	a * b
/	除法	a / b
%	取余	a % b
++	递增	a++ 或 ++a
--	递减	a-- 或 --a

在表 2-4 中，“++”和“--”为一元运算符，可以放在变量前，也可以放在变量后，如++a、a++。虽然这两种运算都是自动+1 的，但是在表达式中是有区别的，其中前一种操作是在 a 参与其他运算前，先加 1；后一种操作是在 a 参与其他运算后，再加 1。

【例 2.13】一元运算符的使用。

```
package jqiang.Symbol
fun main(args: Array<String>) {
    // 一元运算符
    var a=3
    var b=-a
    var c=a
    println("原始值${a}，负数值${b}，正值${c}，加加${++a}，减减${ --b}")
}
```

运行结果如下：

```
原始值 3，负数值-3，正值 3，加加 4，减减-4
```

在 Kotlin 中，使用多元运算符可以操作两个数或者多个数。

【例 2.14】多元运算符的使用。

```
package jqiang.Symbol
fun main(args: Array<String>) {
    var d=15
    var e=10
    var f=5
    println("加${d + e}，减${d - e}，乘${d * e}，除${d / e}，取余${ d
% e}，加加${++d}，减减${ --e}，加减乘${d+e*f}")
}
```

运行结果如下：

```
加 25，减 5，乘 150，除 1，取余 5，加加 16，减减 9 ，加减乘 65
```

2.4.2 关系运算符

关系运算符用于对两个表达式进行比较，返回布尔类型的结果 true 或 false。关系运算符一般在条件或循环语句中起着重要的控制作用。Kotlin 中的关系运算符如表 2-5 所示。

表 2-5

关系运算符	说　明	示　例
<	小于	a < b
>	大于	a > b
<=	小于或等于	a <= b
>=	大于或等于	a >= b
==	等于	a == b

续表

关系运算符	说　明	示　例
!=	不等于	a != b
===	恒等	a === b
!==	非恒等	a !== b

【例 2.15】使用关系运算符对变量中的值进行比较。设置变量 a=10，变量的类型为 Int，将变量 a 与数字 10 进行比较。println 是 Kotlin 中的系统函数，其作用是换行输出变量的相关信息。

```
    package jqiang.Symbol
    fun main(args: Array<String>) {
    // 使用关系运算符，结果返回的是 true 或 false
        val a=10
        println("小于${ a<10 },大于${ a>10 },大于或等于${a >=10 },小于或等于${a <= 10},等于${a == 10},不等于${a != 10},恒等${a === 10},非恒等${a !== 10}")
    }
```

运行结果如下：

```
    小于 false,大于 false,大于或等于 true,小于或等于 true,等于 true,不等于 false,恒等 true,非恒等 false
```

2.4.3　逻辑运算符

逻辑运算符用于连接关系表达式，对关系表达式的值进行逻辑运算。虽然逻辑运算符&&对应于 and，||对应于 or，但是它们的优先级不同。Kotlin 中的逻辑运算符如表 2-6 所示。

表 2-6

逻辑运算符	示　例	结果为真
&& 或 and	a && b 或 a and b	a 和 b 都为真
|| 或 or	a || b 或 a or b	a 和 b 至少有一个为真
xor（逻辑异或）	a xor b	a 和 b 为一真一假
!（逻辑非）	!a	a 为假

【例 2.16】定义两个布尔类型的变量，通过逻辑运算符进行判断。

```
    package jqiang.Symbol
    fun main(args: Array<String>) {
```

```
        var a = true
        var b = false
        println("逻辑或${q || r}, 逻辑或${q or r}, 逻辑与${q && r},逻辑与
${q and r},逻辑异或${q xor r},逻辑非${!r}")
    }
```

运行结果如下：

```
    逻辑或 true, 逻辑或 true, 逻辑与 false,逻辑与 false,逻辑异或 true,逻辑非
false
```

2.4.4 赋值运算符

“=”符号是最简单的赋值运算符，其左边是变量，右边是条件表达式。表达式结果的类型和左边变量的类型一致，或者能转换成左边变量的数据类型。

赋值运算符可与二元算术运算符、逻辑运算符和位运算符组合成简捷运算符，从而简化一些常用表达式的写法。Kotlin 中的赋值运算符如表 2-7 所示。

表 2-7

赋值运算符	说　明	示　例	展开形式	说　明
=	赋值	a = b	a = b	将右边的值赋给左边
+=	加	a += b	a =a + b	将右边的加到左边
-=	减	a -= b	a =a - b	将右边的减到左边
*=	乘	a *= b	a =a * b	将右边的值乘到左边
/=	除	a /= b	a =a / b	将右边的值除到左边
%	取余	a %= b	a =a % b	将左边的值对右边取余数

【例 2.17】通过赋值运算符对两个变量进行重新赋值运算。

```
    package jqiang.Symbol
    fun main(args: Array<String>) {
        // 赋值运算符
        var m=20
        val n=15
        m += n // m = m + n
        // m -= n // m = m - n
        // m *= n // m = m * n
        // m /= n // m = m */n
        // m %=n // m = m % n
        println("${m}")
    }
```

运行结果如下：

```
35
5
300
5
```

2.4.5　位运算函数

虽然 Kotlin 没有像其他编程语言一样提供位运算符，但是在 Kotlin 编程中位运算同样重要，所以 Kotlin 官方提供了特定的函数对 Int 和 Long 类型的数据进行位运算。Kotlin 中提供的位运算函数如表 2-8 所示。

表 2-8

位运算函数	示　例	说　明
shl(bits)	a.shl(2)	有符号左移
shr(bits)	a.shr(2)	有符号右移
ushr(bits)	a.ushr(2)	无符号右移
and(bits)	a.and(2)	位与
or(bits)	a.or(2)	位或
xor(bits)	a.xor(2)	位异或
Inv()	a.inv()	位非

【例 2.18】通过位运算函数计算变量的值。

```
       package jqiang.Symbol
       fun main(args:  Array<String>) {
           val a=10
   println("shl: ${a.shl(2)},shr: ${a.shr(1)},ushr: ${a.ushr(1)},and:
${a.and(1)},or: ${a.or(1)},xor: ${a.xor(1)},inv: ${a.inv()}")
       }
```

运行结果如下：

```
shl: 40,shr: 5,ushr: 5,and: 0,or: 11,xor: 11,inv: -11
```

2.4.6　运算符优先级

所谓的运算符优先级，是指在表达式中哪一种运算符先进行计算，哪一种运算符后进行计算，与数学中的四则运算遵循“先乘除，后加减”的道理一样。

Kotlin 中的运算遵循的规则是：优先级高的操作先执行，优先级低的操作后执行，处于同一个优先级的操作是按照从左到右的顺序进行的。当然，也可以使用小括号，括号内的操作优先进行。Kotlin 运算符优先级如表 2-9 所示。

表 2-9

优 先 级	运 算 符			
1	.	[]	()	++/--
2	++/--	!		
3	*	/	%	
4	+	-		
5	sh1()	shr()	ushr()	
6	>	<	<=	>=
7	==	!=		
8	&			
9	^			
10	|			
11	&&			
12	||			
13	=	+=	-=	
14	*=	/=	%=	

对于运算符的优先级，一时半刻记不住也是可以理解的，所以，在写表达式时，可以使用小括号，例如：a and ((b != c) or 50*(50-d))，这样就减少了很多不必要的麻烦。

2.5 Kotlin 流程控制语句

程序流程控制结构分为：顺序、选择、循环和异常处理。语句是程序的基本组成单位，在 Kotlin 中分为简单语句和复合语句，简单语句就是一行代码，如 var a=10；复合语句就是一些简单的语句组合，如一个方法等。一般语句执行是按照顺序进行的，但是当遇到特殊条件比如循环时，就会按照流程控制执行。

2.5.1　条件语句（if 和 when）

条件语句，是指在程序中根据条件是否成立选择执行的一类语句。条件语句在实际使用中的难点在于如何准确地抽象条件。例如，实现程序登录功能时，如果用户名和密码正确，则进入系统；否则弹出“密码错误”提示框等。

【例 2.19】登录业务逻辑，获取用户基本信息（定义已知变量），根据所获取的信息查询数组中是否存在该用户信息（将用户保存在数组中），如果存在该用户，则返回用户基本信息；如果不存在，则输出不存在该用户的信息。

```
package jqiang.Branch
fun main(args: Array<String>) {
val getuname: String="Kotlin"
val getpwd: String="password"
var getstatus: Boolean ?=true
var db= arrayListOf("Kotlin", "password")
if (getuname!=db[0]) println("用户名不存在")
if (getpwd!=db[1]) println("密码错误")
if (getuname==db[0] && getpwd==db[1]){
    if (db.get(2) == -1){
        println("该用户已删除")
    }else if (db[2]==0){
        println("该用户被禁用")
    }else{
        var str= if (getstatus!!) 1 else 0
        if (str==1){
            println("登录成功，下次进入自动登录状态")
        }else{
            println("登录成功，下次登录请重新输入账户和密码")
        }
    }
}
}
```

运行结果如下：

```
登录成功，下次进入自动登录状态
```

if 语句用于进行条件判断，可以根据所返回的布尔类型的值执行相应的语句块。if 语句分为如下三种形式。

（1）最简单的条件语句是只有一个 if 语句，在程序中根据布尔类型的值进行判断，继续执行或者不执行。if 语句的语法格式为：

```
if(表达式)语句块
```

（2）在程序设计中，如果满足条件就执行一个语句块，不满足条件则执行另一个语句块，此时需要用到 if...else...语句。if...else...语句的语法格式为：

```
if(表达式){
    语句块 1
}else{
    语句块 2
}
```

当条件表达式为 true 时，执行语句块 1；否则，执行语句块 2。

（3）if 语句还有一种形式，比如在判断会员状态时需要用到多个条件，所以可以是多条件语句。多条件语句的语法格式如下：

```
if(表达式 1){
    语句块 1
}else if(表达式 2){
    语句块 2
}else{
    语句块 3
}
```

如果表达 1 成立，则执行语句块 1；如果表达式 1 不成立，则会判断条件是否满足表达式 2，如果满足就执行语句块 2；如果都不满足就执行语句块 3。

以上三种 if 语句可以嵌套使用。Kotlin 还提供了 when 条件语句，与其他语言中的 switch 语句类似，但是它更方便使用。

【例 2.20】使用 when 语句来判断状态值。

```
when(db[2]){
       -1->{ println("该用户已删除") }
    0->{ println("该用户被禁用") }
    in 0..10->{
       var str= if (getstatus!!) 1 else 0
       if (str==1){
          println("登录成功，下次进入自动登录状态")
       }else{
          println("登录成功，下次登录请重新输入账户和密码")
       }
    }else{
       println("登录延迟，请稍后再试")
    }
}
```

在上述例子中，将 when 的参数和所有的分支条件进行顺序比较，直到某个分支满足条件。when 既可以被当作表达式使用，也可以被当作语句使用。如果被当作表达式使用，那么符合条件的分支的值就是整个表达式的值；如果被当作语句使用，则忽略个别分支的值。

如果各分支条件都不满足，则会求 else 分支的值。如果 when 被当作表达式使用，那么必须有 else 分支，除非编译器能够检测出所有可能情况都已经被覆盖。

可以使用任意的表达式作为分支条件，比如函数（parseInt(s)）、区间（in 1..10）、判断数据类型（is String）等。

when 也可以用来取代 if…elseif…语句。如果不提供参数，那以所有的分支条件都是简单的布尔类型条件表达式，当一个分支条件为真时则执行该分支。示例代码如下：

```
when {
    getuname=="Kotlin" -> println("用户名正确")
    else -> println("用户名不存在")
}
```

2.5.2　循环语句

循环语句用于反复执行一段代码，直到满足某个条件为止。在 Kotlin 中有三种循环语句：for 循环语句、while 循环语句和 do...while 循环语句。

当一个循环不知该执行多少次时，可使用 while 或 do...while 循环语句。对于执行已知循环次数的循环，可使用 for 循环语句。一个完整的循环结构包含如下 4 个部分。

（1）初始化部分：用于设置循环的初始化条件，如设置计时器的初始值。

（2）判断部分：一个关系表达式或布尔表达式，用于判断是否可以继续执行循环体。

（3）迭代部分：修改循环初始化条件，控制循环次数，如计时器的值是自增还是自减的。

（4）循环体：在循环中反复执行的代码。

1．for 循环语句

在 Kotlin 中，for 循环语句可以对任意迭代器对象进行遍历，一般表示形式为 for(item in collection){循环体}。

【例 2.21】通过 for 循环语句输出数组中的元素。

```
package jqiang.Branch
fun main(args: Array<String>) {
    // 循环输出 value 值
    val arrayOfString: Array<String> = arrayOf("我", "爱", "Kotlin")
        for (string in arrayOfString) {
        println(string )
    }
    //重复执行
    for (hh in 1..2){
        println("重要的事说 5 遍")
    }
for ((index,value) in arrayOfString.withIndex()) {
    println("${index},${value}")
}

//循环输出数组 index（索引值）
for(index in arrayOfString.indices){
    println(index)
}

}
```

运行结果如下：

```
我
爱
Kotlin
重要的事说 5 遍
重要的事说 5 遍
    0,我
    1,爱
    2,Kotlin
    0
    1
    2
```

2．while 循环

当 while 循环体只有一条语句时，可以将大括号省去。在 while 循环语句中只有一个判断条件，它可以是任何表达式。当判断条件表达式为真时，执行循环体，直到表达式为假跳出循环体。

（1）第一次进入 while 循环前，必须为循环控制变量（或表达式）赋初值。

（2）根据判断条件表达式的值决定是否继续执行循环，如果判断条件表达式的值为真（true），则继续执行循环语句；否则，跳出循环，执行其他语句。

（3）执行完循环体内的语句后，重新为循环控制变量（或表达式）赋值。由于 while 循环语句不会自行更改循环控制变量（或表达式）的值，所以为循环控制变量赋值的工作就要由设计者来做，完成后再回到第 2 步重新判断是否继续执行循环语句。

while 循环一般用在不知道循环次数的情况下，维持循环的是一个条件表达式，条件成立则执行循环体，条件不成立则退出循环。

【例 2.22】计算 1 至 2017 的奇数和。

```
package jqiang.Branch
fun main(args: Array<String>) {
    var nub=1 //起始值
    var times=0 //次数
    var sum=0 //目标值
    while (num<=s 2017){
        if(num%2!=0){
            sum += nub
        }
    }
    println(sum)
}
```

运行结果如下：

```
1018081
```

3．do…while 循环

do...while 循环语句执行的过程是：先执行一次循环体，然后再判断条件表达式，如果条件表达式的值为 true，则继续执行；否则跳出循环。也就是说，do...while 循环中的循环体至少被执行一次，这是与 while 循环不同的地方。

do…while 循环语句也称为后测试循环语句，其执行过程是：先执行一次循环体，然后再判断条件表达式，如果条件表达式的值为 true，则继续执行；否则跳出循环。也就是说，do...while 循环语句中的循环体至少被执行一次。

【例 2.23】使用 do...while 循环语句来实现计算 1~100 的所有整数之和。

```
package jqiang.Branch
fun main(args: Array<String>) {
```

```
    var limit=100
    var sum =0
    var i=1
    do {
        sum+=i
        i++
    }while (i<limit)
    println(sum)
}
```

运行结果如下：

```
5050
```

2.6 跳转语句

在 Kotlin 中有三种跳转语句，分别是 return、break 和 continue。其中 return 可以使用在程序的任何地方；break 和 continue 需要与循环语句一起使用。

1．return

return 语句表示从被调函数返回到主调函数继续执行，返回时可附带一个返回值，由 return 后面的参数指定。return 是必需的，因为在调用函数时其计算结果通常通过 return 返回。即使执行函数不需要返回计算结果，也需要返回一个状态码来表示函数执行是否顺利（-1 和 0 就是最常用的状态码），主调函数可以通过返回值了解被调函数的执行情况。

一般形式为：return 表达式，表达式代表函数的返回值。

【例 2.24】通过函数求和，使用 return 语句。

```
package jqiang.Jump
    fun allSum(vararg x: Int): Int{
    var sum = 0
    for (i in x){
        sum += i
    }
return sum
}
    fun twoSum(x: Int, y: Int): Int{
    return x + y
}
```

```
fun main(args: Array<String>) {
    println("twoSum 函数使用返回的结果"+twoSum(10, 15))
    var a= intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 100)
    println("allSum 函数使用返回的结果"+allSum(*a))
}
```

运行结果如下：

```
twoSum 函数使用返回的结果 25
allSum 函数使用返回的结果 145
```

2．break

break 语句用于结束整个循环，跳出循环，执行后面的语句。

【例 2.25】在输出数组的过程中，当达到某个条件时，中止循环。

```
package jqiang.Jump
fun main(args: Array<String>) {
    var arrs= intArrayOf(1, 2, 3, 4, 5)
    for (arr in arrs) {
        if(arr==3)break
        println(arr)
    }
}
```

运行结果如下：

```
1
2
```

3．continue

continue 语句用于结束本次循环，执行下一次循环。

【例 2.26】在输出数组的过程中，当达到某个条件时，执行下一次循环。

```
package jqiang.Jump
fun main(args: Array<String>) {
    var arrs= intArrayOf(1, 2, 3, 4, 5)
    for (arr in arrs) {
    if(arr==3) continue
        println(arr)
    }
}
```

运行结果如下：

```
1
2
4
5
```

2.7 本章小结

本章通过简单的实例讲解了 Kotlin 语法规范、常量和变量、数据类型、运算符、流程控制语句和跳转语句。其中常量和变量在任何一种编程语言中都是至关重要的，它们是贯穿整个程序的核心，也是编程的基础，其作用就是用于存放数据；流程控制语句在程序中也是必不可少的，无论是入门的数学计算公式，还是高级的复杂算法，都是通过流程控制语句实现的。读者在学习本章内容时，要不断地进行练习和总结，掌握一套属于自己的方法和技巧。

第3章

Kotlin 集合

集合是对大量数据进行有效组织和管理的手段之一，通过集合的强大功能，可以对大量的数据进行相同的数据存储、排序、插入及删除等操作，从而有效提高程序开发效率。虽然 Kotlin 进入市场比较晚，但是它凭借着代码开源、升级快等特点，对程序有一套更加完善的体系。Kotlin 和其他编程语言有点不一样，它对集合掌控得非常严格，可以创建只读集合和可变集合。

集合（简称“集”），是数学中的一个基本概念，它是集合论的研究对象，集合论的基本理论直到 19 世纪才被创立。集合是“确定的一堆东西”，集合里的“东西”叫元素。

在数学中，由一个或多个确定元素构成的整体叫集合。若 x 是集合 A 的元素，则记作 $x \in A$。集合中的元素有三个特征：①确定性，集合中的元素必须是确定的；②互异性，集合中的元素互不相同，如集合 $A=\{1, a\}$，则 a 不能等于 1；③无序性，集合中的元素没有先后之分，如集合{3,4,5}和{3,5,4}算作同一个集合。

本章主要内容：

- 集合的简单介绍
- 集合 List：只读集合和可变集合的创建及使用
- 集合 Set：只读集合和可变集合的创建及使用
- 集合 Map：只读集合和可变集合的创建及使用
- 元素操作符

3.1 集合

不同于 Java 集合，Kotlin 集合根据“是否可变”分为两种：可变集合和不可变集合。不可变集合是在可变集合的前面加上 Mutable——列表：List/MutableList；集：Set/MutableSet；映射：Map/MutableMap；集：Collection/MutableCollection；迭代器：Iterable/MutableIterable。

应用程序开发离不开数据，进行数据处理就会用到集合，只有深入了解不同类型的集合分别实现了哪些方法，才能在需要时快速选出最合适的集合与对应的方法。

在 Kotlin 中集合主要有以下几种。

- Iterable：可表示为一系列元素的集合或另一个实体上的迭代器。
- MutableIterable：可变集合上的迭代器。提供在迭代时移除元素的能力。
- Collection：元素的通用集合。此接口中的方法仅支持对集合的只读访问。
- MutableCollection：支持添加和删除集合中的元素。
- List：元素的通用有序集合。此接口中的方法仅支持对列表的只读访问。
- MutableList：支持添加和删除集合中的元素。
- Set：不支持重复元素的通用无序集合。
- MutableSet：不支持重复元素的一般无序集合，支持添加和删除元素。
- Map：集合中保存对象（键和值），并支持有效地检索与每个键对应的值。Map 集合中的键值是独一无二的，每个键都对应一个值。
- MutableMap：一个可修改的集合，可以保存对象（键和值），并支持有效地检索与每个键对应的值。Map 集合中的键值是独一无二的，每个键都对应一个值。

3.2 集合之 List

List 是一个元素有序并可以重复的集合，在 Kotlin 中可以创建只读集合和可变集合。

创建 List 集合的一般形式如下：

```
val emptylist= emptyList<String>()//创建一个空的 List 集合
```

```
val list = listOf<Int>(1,2,3)//创建一个只读的 List 集合
val mlist= mutableListOf<Int>(1,2,3)//创建一个可变的 List 集合
```

1．只读 List 集合

创建只读的 List 集合时继承 Collection<E>接口，元素以线性方式存储，在集合中可以存放重复的对象。

listOf()是使用 ArrayList 实现的，返回的 List 集合是只读的。在程序开发中，应尽量使用只读 List 集合，因为这在一定程度上可以提高内存效率。

在 Kotlin 中，对 List 集合常见的 API 操作如下：

（1）修改操作

- val size: Int——统计集合中元素的数量。
- fun isEmpty(): Boolean——判断集合是否为空。
- fun contains(E): Boolean——判断集合中是否包含某一元素。
- fun iterator(): Iterator——返回该只读集合中元素的迭代器。

（2）批量操作

- fun containsAll(Collection<E>): Boolean——判断集合中是否包含某一集合。

（3）索引操作

- fun get(Int): E——查询集合中某个位置的元素。

（4）搜索操作

- fun indexOf(E): Int——返回列表中指定元素首次出现的索引值，如果元素不包含在列表中，则返回-1。
- fun lastIndexOf(E): Int——返回列表中指定元素最后一次出现的索引值，如果元素不包含在列表中，则返回-1。

（5）迭代器

- fun listIterator(): ListIterator——返回一个集合的迭代器。
- fun listIterator(Int): ListIterator——从指定位置开始，返回集合的迭代器。
- fun subList(fromIndex: Int, toIndex: Int): List——返回列表中指定区间的集合

Kotlin 并没有提供创建 List 集合的函数，如果想创建 List 集合，可以调用标准库中的 listOf()和 mutableListOf()方法。

【例 3.1】使用 listof()创建 List 集合，并输出集合。

```
package jqiang.sets
fun  main(args: Array<String>) {
   val device1= listOf("显示器","键盘","鼠标","主机")
   val device2= listOf("联想笔记本","戴尔笔记本","外星人笔记本")
```

```
        val devices= listOf(device1,device2)
        println("取出 index: ${devices.indexOf(device1)}的 value 值:
${devices.get(1)}取出最后一次出现的位置${devices.lastIndexOf(device2)}")
        //循环输出
        devices.forEach(::println)
        for (device in devices) {
            println(device)
        }
    }
```

运行结果如下：

```
    取出 index：0 的 value 值：[联想笔记本, 戴尔笔记本, 外星人笔记本]取出最后一次
出现的位置 1
    [显示器, 键盘, 鼠标, 主机]
    [联想笔记本, 戴尔笔记本, 外星人笔记本]
    [显示器, 键盘, 鼠标, 主机]
    [联想笔记本, 戴尔笔记本, 外星人笔记本]
```

2．可变 List 集合

MutableList<E>接口继承自 List<E>，MutableCollection 是对只读集合的扩展，可以对集合元素进行添加和删除操作。

在 Kotlin 中，对 MutableList 集合常见的 API 操作如下：

（1）修改操作

- fun add(E): Boolean——向集合中添加元素。如果添加成功，则返回 true；否则返回 false。
- fun remove(E): Boolean——移除集合中的元素。如果移除成功，则返回 true；否则返回 false。

（2）批量操作

- fun addAll(Collection): Boolean——向集合中添加一个集合。如果添加成功，则返回 true；否则返回 false。
- fun removeAll(Collection): Boolean——移除集合中的一个集合。如果移除成功，则返回 true；否则返回 false。
- retainAll(Collection): Boolean——判断集合中是否包含一个集合。如果包含，则返回 true；否则返回 false。
- fun clear(): Unit——将集合中的元素清空。

（3）索引操作

- fun set(Int, E): E——用指定的元素替换列表中指定位置的元素，并返回该

位置原来的元素。

- fun add(Int, E): Unit——在指定位置添加元素。
- fun removeAt(Int): E——移除指定索引处的元素。

（4）迭代器

- fun listIterator(): MutableListIterator——返回一个集合的迭代器。
- fun listIterator(Int): MutableListIterator——从指定位置开始，返回集合的迭代器。
- fun subList(fromIndex: Int, toIndex: Int): MutableList——返回列表中指定区间的集合。

3．扩展函数 toList

toList 扩展函数用于复制 List 中的内容，因此所返回的 List 集合永远不会改变，通过 toList 函数处理后生成一个只读的 List 集合。

3.3　集合之 Set

Set 是最简单的一种集合，Set 集合中的对象不按特定方式排序，并且没有重复对象。

Kotlin 没有提供专门的语法用来创建 Set 集合，可以使用标准库中的 setOf()和 mutableSetOf()方法。

创建 Set 集合的一般形式如下：

```
val emptyset= emptySet<String>()//创建一个空的 Set 集合
val set= setOf(1,2,3)//创建一个只读的 Set 集合
val mset= mutableSetOf(1,2,3)//创建一个可变的 Set 集合
val hasset= hashSetOf(1,2,3)//创建一个 hashset 集合
val linkset= linkedSetOf(1,2,3)//创建一个 linkedset 集合
val sortset= sortedSetOf(1,2,3)//创建一个 sortedset 集合
```

1．只读 Set 集合

在 Kotlin 中，对 Set 集合常见的 API 操作如下：

（1）查询操作

- val size: Int——返回集合中元素的数量。
- fun isEmpty(): Boolean——判断集合是否为空。

- fun contains(E): Boolean——判断集合中是否包含某一元素。
- fun iterator(): Iterator——返回只读集合中元素的迭代器。

（2）批量操作

- fun containsAll(Collection<E>): Boolean——判断集合中是否包含某一集合。

【例 3.2】创建一个只读的 Set 集合。

```
package jqiang.sets
fun main(args: Array<String>) {
    val sets= setOf(1,2,3,1,3,2,2,3,1)//创建一个只读的 Set 集合
    sets.forEach(::println)
}
```

运行结果如下：

```
1
2
3
```

从上面示例中可以清楚地看出，Set 集合中没有重复对象，并且对象不按特定方式排序。

2．可变 Set 集合

MutableSet 接口继承自 Set，MutableCollection 是对 Set 集合的扩展，可以对集合元素进行添加和删除操作。

在 Kotlin 中，对 MutableSet 集合常见的 API 操作如下：

（1）查询操作

- fun iterator(): MutableIterator——返回集合元素的迭代器。

（2）元素操作

- fun add(E): Boolean——向集合中添加元素。如果添加成功，则返回 true；否则返回 false。
- fun remove(E): Boolean——移除集合中的元素。如果移除成功，则返回 true；否则返回 false。

（3）批量操作

- fun addAll(Collection): Boolean——向集合中添加一个集合。如果添加成功，则返回 true；否则返回 false。
- fun removeAll(Collection): Boolean——移除集合中的一个集合。如果移除成功，则返回 true；否则返回 false。

- retainAll(Collection): Boolean——判断集合中是否包含一个集合。如果包含，则返回 true；否则返回 false。
- fun clear(): Unit——将集合中的元素清空。

【例 3.3】创建一个可变的 Set 集合。

```
package jqiang.sets
fun main(args: Array<String>) {
    val mset= mutableSetOf(1,2,3)//创建一个可变的 Set 集合
    println(mset)
    mset.add(4)
    println(mset)
    mset.remove(2)
    println(mset)
}
```

运行结果如下：

```
[1, 2, 3]
[1, 2, 3, 4]
[1, 3, 4]
```

3.4　集合之 Map

Map 是一种将键映射到值的对象，一个映射不能包含重复的键；每个键最多只能映射到一个值。Map 没有继承自 Collection 接口，从 Map 集合中检索元素时，只要给出键对象，就会返回对应的值对象。

在 Kotlin 中，与 List、Set 集合一样，Map 集合也分为只读 Map 集合和可变 Map 集合。创建 Map 集合时，需要调用标准库中的 mapOf()和 mutableMapOf()方法。

创建 Map 集合的一般形式如下：

```
    val emptyMap= emptyMap<Int,String>()//创建一个空的 Map 集合
    val map= mapOf(1 to "one",2 to "two",3 to "three")//创建一个普通的
Map 集合
    val mmap= mutableMapOf(1 to "one",2 to "two",3 to "three")//创建一
个可变的 Map 集合
    val hashmap= hashMapOf(1 to "one",2 to "two",3 to "three")//创建一
个 hashmap 集合
    val sortedmap= sortedMapOf(1 to "one",2 to "two",3 to "three")//
创建一个 sortedmap 集合
```

HashMap：Map 集合基于散列表的实现。插入和查询“键值对”的开销是固定的。可以通过构造器设置容量（capacity）和负载因子（load factor），以调整容器的性能。

LinkedHashMap：类似于 HashMap，但是迭代遍历它时，所取得的“键值对”的顺序是其插入次序，或者是最近最少使用（LRU）的次序。LinkedHashMap 在迭代访问时的速度比 HashMap 更快，因为它使用链表维护内部次序。

通过 mapOf()和 mutableMapOf()创建的 Map 是基于 Java 的 LinkedHashMap。

目前 Kotlin 并不支持 TreeMap、WeakHashMao 和 IdentifyHashMap。

【例 3.4】创建两个 Map 集合，并对集合进行常规操作。

```
   package jqiang.sets
   package jqiang.sets

   fun main(args: Array<String>) {
      val map= mapOf(1 to "one",2 to "two",3 to "three")// 创建一个普通的 Map 集合
      val mmap= mutableMapOf(1 to "one",2 to "two",3 to "three")//创建一个可变的 Map 集合

      println("映射中元素的数量"+map.size)
      println("Map 元素中 key 的集合"+map.keys.toList())
      println("Map 元素中 value 的集合"+map.values.toList())
      println(map.entries)
      println(map.get(1))
      map.forEach { key, value -> println("Map 中元素的键值对的集合 key: ${key} , value: ${value}") }
      map.forEach{ println("Map 中元素的键值对的集合 key: ${it.key} , value: ${it.value}") }
      if (!map.isEmpty()){
         println("Map 集合不为空")
      }
      if (map.containsKey(1)){
         println("Map 包含 key 值为 1")
      }
      if (map.containsValue("one")){
         println("Map 包含 value 值为 one")
      }
      println(mmap)
       mmap.put(4,"four")
      println(mmap)
      val addall= mapOf(5 to "five",6 to "six")
```

```
    mmap.putAll(addall)
    println(mmap)
    mmap.remove(1)
    println(mmap)
    mmap.clear()
    println(mmap)

}
```

运行结果如下：

```
映射中元素的数量 3
Map 元素中 key 的集合[1, 2, 3]
Map 元素中 value 的集合[one, two, three]
[1=one, 2=two, 3=three]
one
Map 中元素的键值对的集合 key: 1 , value: one
Map 中元素的键值对的集合 key: 2 , value: two
Map 中元素的键值对的集合 key: 3 , value: three
Map 中元素的键值对的集合 key: 1 , value: one
Map 中元素的键值对的集合 key: 2 , value: two
Map 中元素的键值对的集合 key: 3 , value: three
Map 集合不为空
Map 包含 key 值为 1
Map 包含 value 值为 one
{1=one, 2=two, 3=three}
{1=one, 2=two, 3=three, 4=four}
{1=one, 2=two, 3=three, 4=four, 5=five, 6=six}
{2=two, 3=three, 4=four, 5=five, 6=six}
{}
```

3.5　集合操作符

在 Kotlin 中，关于集合的操作符分为 6 类，分别是：总数操作符、过滤操作符、映射操作符、顺序操作符、生产操作符和元素操作符。

3.5.1　总数操作符

使用总数操作符，对集合进行简单的筛选（满足某条件），以生成新的集合。

在 Kotlin 中常见的总数操作符如下：

- any——判断集合中是否有满足条件的元素。
- all——判断集合中的元素是否都满足条件。
- none——判断集合中的元素是否都不满足条件，是则返回 true。
- count——查询集合中满足条件的元素个数。
- reduce——从第一个元素到最后一个元素进行累计。
- reduceRight——从最后一个元素到第一个元素进行累计。
- fold——与 reduce 类似，有初始值，而不是从 0 开始累计的。
- foldRight——与 reduceRight 类似，有初始值，而不是从 0 开始累计的。
- forEach——循环遍历元素，对于 forEach 本身需要使用 it 关键字对每个元素进行相关操作。
- forEachIndexed——循环遍历元素，同时得到元素索引（下标）。
- max——查询最大的元素，如果没有则返回 null。
- maxBy——获取处理后返回结果最大值对应的那个元素的初始值，如果没有则返回 null。
- min——查询最小的元素，如果没有则返回 null。
- minBy——获取处理后返回结果最小值对应的那个元素的初始值，如果没有则返回 null。
- sumBy——获取处理后返回结果值的总和。
- dropWhile——返回从第一项起，去掉满足条件的元素，直到不满足条件的一项为止。

【例 3.5】创建一个集合，使用总数操作符进行操作。

```
package jqiang.sets
fun main(args: Array<String>) {
    val list= listOf(1,2,3,4,5,6,7,8,9,10)
    println(list.any{it > 10})
    println(list.all{it in 1..10})
    println(list.sumBy { it*it })
    println(list.min ())
    println(list.minBy { it*it })
}
```

运行结果如下：

```
false
true
```

```
385
1
1
```

3.5.2　过滤操作符

使用过滤操作符，根据某个条件对集合中的元素进行过滤。在 Kotlin 中常见的过滤操作符如下：

- filter——过滤掉所有满足条件的元素。
- filterNot——过滤掉所有不满足条件的元素。
- filterNotNull——过滤掉 null。
- take——返回从第一个元素开始的 *n* 个元素。
- takeLast——返回从最后一个元素开始的 *n* 个元素。
- takeWhile——返回从第一个元素开始符合给定函数条件的元素。
- drop——返回去掉前 *n* 个元素后的列表。
- dropLastWhile——返回从最后一项起，去掉满足条件的元素，直到不满足条件的一项为止。
- slice——过滤掉非指定下标对应的元素，即保留指定下标对应的元素。

【例 3.6】创建一个集合，使用过滤操作符进行操作。

```
package jqiang.sets

fun main(args: Array<String>) {
    val list= listOf(1,2,3,4,5,6,7,8,9,10)
    //过滤操作符
    println(list.filter { it !=5 })
    println(list.filterNot { it==5 } )
    println(list.filterNotNull() )
    println(list.take(4) )
    println(list.takeLast(4))
    println(list.takeLastWhile { it !!< 5 } )
    println(list.drop(4) )
    println(list.dropLastWhile { it==4 } )
    println(list.dropWhile { it!!<4 } )
    println(list.slice(listOf(1,2,3)) )
}
```

运行结果如下：

```
[1, 2, 3, 4, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4]
[7, 8, 9, 10]
[]
[5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[4, 5, 6, 7, 8, 9, 10]
[2, 3, 4]
```

3.5.3 映射操作符

对原来的集合进行筛选之后，可以使用映射操作符将原来的集合生成为一个新集合。在 Kotlin 中常见的映射操作符如下：

- map——对集合中的元素通过某方法转换后，将结果保存到一个集合中。
- mapIndexed——除了可以得到转换后的结果，还可以得到索引（下标）。
- mapNotNull——在执行方法转换前过滤掉值为 NULL 的元素。
- flatMap——合并两个集合，可以在合并时做些小动作。
- groupBy——将集合中的元素按照某个条件进行分组，返回 Map。

【例 3.7】使用映射操作符对集合进行操作。

```
package jqiang.sets
fun main(args: Array<String>) {
    val list1= listOf(1,2,3,4,5)
    val list2= listOf(6,7,8,9,10)
    // 映射操作符
    println(list1.map { it+1 })
    println(list1.mapIndexed { index, i -> index*i  })
    println(list1.mapNotNull { it+5 })
    println(listOf(list1,list2).flatMap { it->it })
    println(listOf(list1.groupBy { if (it>3)"big" else "small"}))
}
```

运行结果如下：

```
[2, 3, 4, 5, 6]
[0, 2, 6, 12, 20]
[6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[{small=[1, 2, 3], big=[4, 5]}]
```

3.5.4　顺序操作符

如果集合中元素的顺序很乱，那么可以使用顺序操作符重新生成一个有序的集合。在 Kotlin 中常见的顺序操作符如下：

- reversed——以相反顺序排列。
- sorted——自然排序（升序）。
- sortedBy——根据方法处理结果进行自然（升序）排列。
- sortedDescending——降序排列。
- sortedByDescending——根据方法处理结果进行降序排列。

【例 3.8】使用顺序操作符对集合进行操作。

```
package jqiang.sets
fun main(args: Array<String>) {
    // 顺序操作符
    val list1= listOf(1,5,9,7,26,74,32,47,41,42,6)
    val list2= listOf(11,56,8,4,6,4,2,69,7)
    println(list1.reversed())
    println(list1.sorted())
    println(list1.sortedBy { it % 2  })
    println(list1.sortedDescending())
    println(list1.sortedByDescending { it % 2 })
}
```

运行结果如下：

```
[6, 42, 41, 47, 32, 74, 26, 7, 9, 5, 1]
[1, 5, 6, 7, 9, 26, 32, 41, 42, 47, 74]
[26, 74, 32, 42, 6, 1, 5, 9, 7, 47, 41]
[74, 47, 42, 41, 32, 26, 9, 7, 6, 5, 1]
[1, 5, 9, 7, 47, 41, 26, 74, 32, 42, 6]
```

3.5.5　生产操作符

生产操作符主要用于把两个集合按照相同的下标合成一个新的集合，这个新的集合的大小由最小的集合决定。

在 Kotlin 中常见的生产操作符如下：

- zip——将两个集合按照下标组合成一个个 Pair 放到集合中返回。

```
assertEquals(
    listOf(Pair(1, 7), Pair(2, 8)),
```

```
    list.zip(listOf(7, 8))
)
```

- partition——根据判断条件是否成立，拆分成两个 Pair。

```
assertEquals(
    Pair(listOf(2, 4, 6), listOf(1, 3, 5)),
    list.partition { it % 2 == 0 }
)
```

- plus——合并两个 List 集合。可以用“+”替代 plus。
- unzip——将包含多个 Pair 的 List 集合转换成含 List 集合的 Pair。

```
assertEquals(
    Pair(listOf(5, 6), listOf(7, 8)),
    listOf(Pair(5, 7), Pair(6, 8)).unzip()
)
```

【例 3.9】使用生产操作符对集合进行操作。

```
package jqiang.sets
fun main(args: Array<String>) {
    val list1= listOf(1,2,3,4,5)
    val list2= listOf(6,7,8,9,10,11)
    // 生产操作符
    println(list1.zip(list2))
    println(list1.zip(list2){it1,it2->it1+it2})
    println(list1.partition { it > 3 })
    println(list1.plus(list2))
    println(listOf(Pair(1,2),Pair(3,4)).unzip())
}
```

运行结果如下：

```
[(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
[7, 9, 11, 13, 15]
([4, 5], [1, 2, 3])
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
([1, 3], [2, 4])
```

3.5.6 元素操作符

在 Kotlin 中常见的元素操作符如下：

- contains——判断集合中是否有指定元素，有则返回 true。

- elementAt——查找下标对应的元素，如果下标越界，则抛出 IndexOutOfBoundsException 异常。
- elementAtOrElse——查找下标对应的元素，如果下标越界，则根据方法返回默认值（最大下标经方法处理后的值）。
- elementAtOrNull——查找下标对应的元素，如果下标越界，则返回 Null。
- first——返回符合条件的第一个元素，如果没有则抛出 NoSuchElementException 异常。
- firstOrNull——返回符合条件的第一个元素，如果没有则返回 Null。
- indexOf——返回指定下标对应的元素，如果没有则返回-1。
- indexOfFirst——返回第一个符合条件的元素下标，如果没有则返回-1。
- indexOfLast——返回最后一个符合条件的元素下标，如果没有则返回-1。
- last——返回符合条件的最后一个元素，如果没有则抛出 NoSuchElementException 异常。
- lastIndexOf——返回符合条件的最后一个元素，如果没有则返回-1。
- lastOrNull——返回符合条件的最后一个元素，如果没有则返回 Null。
- single——返回符合条件的单个元素，如果没有符合条件的或有超过一个的，则抛出异常。
- singleOrNull——返回符合条件的单个元素，如果没有符合条件的或有超过一个的，则返回 Null。

【例 3.10】使用元素操作符对集合进行操作。

```
package jqiang.sets
fun main(args: Array<String>) {
    val list1= listOf(1,2,3,4,5,6,7,8,9,10)
    println(list1.contains(14))
    println(list1.elementAt(4))
    println(list1.elementAtOrElse(14,{it+3}))
    println(list1.elementAtOrNull(14))
    println(list1.first())
    println(list1.first { it % 3 ==0 })
    println(list1.firstOrNull{it > 14})
    println(list1.indexOf(5))
    println(list1.indexOfFirst { it==14 })

    println(list1.lastOrNull { it==8 })
    println(list1.single { it==8 })
    println(list1.singleOrNull { it==8 })
}
```

运行结果如下：

```
false
5
17
null
1
3
null
4
-1
8
8
8
```

3.6 本章小结

本章主要介绍了 Kotlin 中常见的集合（List、Map 和 Set）并细化为可变的和不可变的集合。还介绍了针对这些集合利用集合操作符进行筛选，返回可用的集合。集合在程序中用于存储数据，是很重要的组成部分，希望读者认真领会本章内容，做到灵活运用。

第4章

Kotlin 函数

本章主要介绍在 Kotlin 中如何定义函数、函数的使用方式，以及使用 Lambda 表达式来声明函数，还简单介绍了 Kotlin 中的协程。

在实际工作中，对于大的任务，一般将其分解成多个较小的任务，由多人分工合作完成。编写程序也是如此，比如有的程序规模较大，为了方便多人分工完成，每个程序都包含多个函数，每个函数都是单独编译与测试的，且可以反复使用，这样就可以减少程序员的编写工作量，提高代码编写效率。

本章主要内容：

- 函数的定义
- 函数的调用
- 高阶函数
- 内联函数
- Lambda 表达式
- 协程

4.1 模块化程序设计

模块化是指将大型程序按照功能分解成若干个相对独立的功能模块，然后分别进行设计，最后把这些功能模块再按照层次关系组装起来。

独立模块由顺序、分支和循环这三种基本结构组成，其特点主要表现在主程序与独立模块之间的数据输入和输出，即主程序与模块函数之间的数据传递。

【例 4.1】计算 1!+2!+3!+…+7!+8!的值。

```
package jqiang.function
//累加求和
fun sum( n:Int): Double {
    var i=1
    var s:Double=0.0
    do {
        s=s+ fac(i)
        i++
    }while (i<=n)
    return s
}
// 阶乘
fun fac(n:Int): Double {
    var i:Int=1
    var f:Double=1.0

    for (hh in 1..n){
        f=f*i
        i++

    }
    return f
}
fun main(args: Array<String>) {
    println( sum(8))
}
```

对于上面的实例，整个程序分解为求和与求阶乘两个功能函数。其中 sum()函数完成求和，fac()函数完成求阶乘。

4.2 函数定义

在 Kotlin 中使用 fun 关键字来定义函数，在 fun 后面紧跟着函数名，函数名称必须以字母、数字、下画线以及中文字符开头。

函数定义的一般格式如下：

```
fun 函数名([参数:类型]):类型{
    执行语句 [return 返回值]
}
```

例如：

```
fun double(x: Int): Int {
    return 2*x
}
```

上面定义一个函数，函数名是 double，整型 x 是函数的形参，使用花括号括起来的部分是函数体，return 2*x 是返回语句，函数的返回值是 int 类型。

说明：

（1）函数名：虽然对函数名称没有严格的要求，但是要知名见意，以增强程序的可读性。

（2）参数列表根据需求进行设置，至少有一个参数。如果需要传入多个参数，那么参数之间需要使用逗号隔开，其中传入的参数可以携带默认值和数据类型。

（3）函数体用一对花括号（{}）括起来，其内部主要包含声明和功能部分。

（4）return（表达式）是返回语句，返回值的类型必须与声明函数时的数据类型一致，如果不一致，系统会自动根据返回值的类型进行强制转换，如果不能转换则报错。

（5）在 Kotlin 中，无论有多少个函数，其中必有一个名为 main 的函数，它是函数的执行入口。

（6）各个函数之间的关系是平行的，主函数的地位并不高于被调函数，它与其他函数之间的关系也是平行的，不能说 main 函数的地位高于其他函数。

【例 4.2】无默认值函数。

```
package jqiang.function
fun double(x: Int): Int {
    return 2*x
}
```

【例 4.3】有默认值函数。

```
package jqiang.function
fun plus(a: Int , b: Int=0 ): Int {
    return a + b
}
```

【例 4.4】单表达式函数。

```
    package jqiang.function
fun ex(x: Int): Int = x * 2
```

【例 4.5】如果一个函数不返回任何有意义的结果值，那么它的返回类型为

Unit。Unit 类型只有唯一的一个值 Unit，在函数中不需要明确返回这个值。对于返回值为 Unit 的函数，Unit 可以省略。

```
package jqiang.function
fun no(nub: Int):Int{
}
fun main(args: Array<String>) {
    println(no(10))
}
```

运行结果如下：

```
kotlin.Unit
```

为了在程序项目中避免出现无意义的 unit 值，请多添加一个条件判断，判断该值是否为 null 值，如果为 null 值就返回 0；否则返回该值。

无论是封装函数还是封装一个操作类，都请针对每一个变量进行检查，避免出现 unit 值。

```
fun no(nub: Int): Int {
    if (nub!=null){
        return nub
    }else{
        return 0
    }
}
```

4.3 函数调用

在 Kotlin 中，程序是由函数组成的，函数的执行是通过函数调用完成的。程序中的 main()函数是由操作系统调用执行的，而其他函数都是由 mian()函数直接或间接调用执行的。

任何一个函数都不会主动执行，必须由主调函数调用它来执行。主调函数可以是主函数 main()，也可以是其他函数。一个函数在执行过程中去执行另一个函数（被调函数），称为函数调用。函数调用的一般形式为：函数名([参数列表])。

【例 4.6】无默认值的函数调用。

```
fun main(args: Array<String>) {
    println(double(10))
}
```

运行结果如下：

```
20
```

【例 4.7】有默认值的函数调用。

```
fun main(args: Array<String>) {
    println(plus(10))
    println(plus(10,15))
}
```

运行结果如下：

```
10
25
```

4.4　可变参数函数

当一个函数需要接受多个参数时，即该函数可以传入数组形式的参数，可以使用可变参数函数进行函数封装。

可变参数不必是函数参数列表中的最后一个参数；使用 vararg 关键字声明可变参数；和 Java 一样，在函数内部可以以数组的形式使用这个可变参数的形参变量。

【例 4.8】可变参数函数的使用。

```
package jqiang.function
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts)  result.add(t)
    return result
}
fun main(args: Array<String>) {
    val a = arrayOf(1, 2, 3)
    val list = asList(-1, 0, *a, 4)
    println(list)
}
```

运行结果如下：

```
[-1, 0, 1, 2, 3, 4]
```

在函数内部 T 相当于一个数组，在例 4.8 中 ts 变量是 Array<out T>的引用。

只有一个参数可以用 vararg 标识，如果被 vararg 标识的不是最后一个参数，则可以通过命名参数法来赋值；如果参数是函数类型的，则可以传入 Lambda 表达式。

当调用一个 vararg 函数时，可以一个个传入参数，就像 asList(1,2,3)，或者将一个定义好的数组当作参数传入，可以用“*”前缀来引用。

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

4.5 尾递归函数

函数的递归调用，是指在函数体内直接或间接地调用函数本身。在 Kotlin 中，尾递归函数需要使用 tailrec 修饰符进行标识，并且只有满足所需格式才可调用其本身。

【例 4.9】使用尾递归函数求 *n*！。负数没有阶乘，0 的阶乘是 1，整数的阶乘 *n*！=*n*(*n*-1)!，(*n*-1)!=(*n*-1)(*n*-2)，依此类推。

```
tailrec fun fac(n:Int): Double {
    var t:Double=1.0
    if (n==0||n==1){
        t=1.0
    }else{
        t= n * fac(n-1)
    }
    return t
}
```

符合 tailrec 修饰符的条件：函数必须将其本身调用作为它执行的最后一个操作，在异常处理中不能使用尾递归函数，目前尾递归函数在 JVM 后端使用。

4.6 高阶函数

将函数用作参数或返回值的函数，称作高阶函数。比如 lock()，它接收一个锁对象和一个函数，获取锁，运行函数并释放锁。示例代码如下：

```
fun <T> lock(lock: Lock, body: () -> T): T {
    lock.lock()
    try {
        return body()
    }
    finally {
        lock.unlock()
    }
}
```

如果调用 lock()，则可以传递另一个函数作为参数。

```
fun toBeSynchronized() = sharedResource.operation()
val result = lock(lock, ::toBeSynchronized)
```

还有一种更方便的方式是传递 Lambda 表达式，示例代码如下：

```
val result = lock(lock, { sharedResource.operation() })
```

在 Kotlin 中，如果一个函数的最后一个参数是函数，并且传递一个 Lambda 表达式作为相应的参数。示例代码如下：

```
lock (lock) { sharedResource.operation() }
```

在 Kotlin 中有很多高阶函数供开发者使用，常见的高阶函数有 apply()、lazy()、let()、repeat()、run()、to()、with()等。

ma()是常见的高阶函数，示例代码如下：

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
        for (item in this)
            result.add(transform(item))
    return result
}
```

map()函数的使用方式如下：

```
val doubled = ints.map { value -> value * 2 }
```

注意：括号可以被完全忽略，当参数是一个简单的表达式时，可以省略声明部分，直接使用 it 关键字访问元素本身。

4.7 内联函数

实际上，调用某个函数是将程序执行顺序转移到内存中存放该函数的地址，该函数执行完后，再返回执行该函数前面的代码。这种转移操作要求在转去前要保护现场并记忆执行的地址，转回后要恢复现场，并按原来保存的地址继续执行。这就是所谓的压栈和出栈。因此，函数调用会有一定时间和空间的开销，对于那些函数体不是很大，但是却频繁调用的函数来说，这个时间和空间的开销会很大。

解决性能消耗问题需要引入内联函数，在编译程序时，编译器将程序中出现的内联函数调用表达式用它的函数体直接进行替换，显然这样不会出现转来转去的情况。因为在 Kotlin 编译过程中函数体中的代码会被替换，提供给主函数运行，因此会增加目标程序的代码量，进而增加空间开销；而在时间开销上，没有函数调用时大，可见它是以目标代码的增加为代价来换取时间的。

在 Kotlin 中，使用 inline 修饰符标识内联函数。使用内联函数既会影响函数本身，也会影响传递给它的 Lambda 表达式，这两者都会被内联到调用处。代码如下：

```
inline fun lock<T>(lock: Lock, body: () -> T): T { // ... }
```

编译器可直接产生如下代码，不必为参数创建函数对象后，再调用这个参数指向的函数。

```
l.lock() try { foo() } finally { l.unlock() }
```

假如在一个内联函数的参数中有多个 Lambda 表达式，如果只希望内联其中的一部分，则可以对函数的部分参数添加 noinline 修饰符。代码如下：

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit)
{ // ... }
```

内联的 Lambda 表达式只能在内联函数内部调用，或者作为可内联的参数传递给其他函数，但使用 noinline 标识的 Lambda 表达式可以按照自己喜欢的方式任意使用，比如可以保存在域内，也可以当作参数传递等。

在 Kotlin 中，我们可以不加条件地使用 return 退出一个命名函数或表达式函数。这意味着退出一个 Lambda 函数时，不得不使用标签，而且无法返回值的 return 在 Lambda 函数中是禁止的，因为 Lambda 函数不可以是一个闭合函数。示例代码如下：

```
    fun foo() { ordinaryFunction { return // 错误：这里不允许让 `foo` 函
数返回 } }
```

如果 Lambda 表达式被传递过去的函数是内联函数，那么 return 语句也可以内联，因此允许使用 return。示例代码如下：

```
    fun foo() { inlineFunction { return // 正确：这里的 Lambda 表达式是内
联的 } }
```

有些内联函数可能并不在自己的函数体内直接调用传递给它的 Lambda 表达式参数，而是通过另一个执行环境来调用，如通过一个局部对象或者一个嵌套函数来调用。在这种情况下，在 Lambda 函数中也不允许存在非局部的控制流。在 Kotlin 中为了使 Lambda 表达式不被禁止，可以使用 crossinline 关键字修饰 Lambda 表达式的参数，或者使用 InlineOptions.ONLY_LOCAL_RETURN 注解。示例代码如下：

```
    inline fun f(inlineOptions(InlineOption.ONLY_LOCAL_RETURN) body: ()
-> Unit) {
        val f = object: Runnable {
        override fun run() = body()
        }
    }
```

4.8　Lambda 表达式

函数式编程依赖于函数和不变性，这样调用一个函数时始终会返回相同的结果。通常，折中即是完美，所以现在大多数开发语言（如 Kotlin 和 Scala）中都融合了过程式编程和函数式编程两种方式，并在这两方面拥有最精彩的灵感和创意。有些问题使用函数式编程方式解决更好，而使用过程式编程方式解决则更直接。

Lambda 表达式是定义匿名函数的简单形式，Lambda 表达式避免了在抽象类和接口中编写明确的函数声明，进而也避免了类的实现部分，因此十分有用。在 Kotlin 中，可以将一个函数作为另一个函数的参数，示例代码如下：

```
    val sum = { x: Int, y: Int -> x + y }
```

Lambda 表达式总是写在大括号内，在完整的语法形式中，将参数声明放在括号内，并有可选的类型标注，函数体位于“->”符号之后。如果推断出该 Lambda 表达式的返回类型不是 Unit，那么该 Lambda 表达式中的最后一个（也可能是单

个）表达式会视为返回值。

把所有的可选标注都留下，示例代码如下：

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

一个 Lambda 表达式只有一个参数是很常见的，函数文本有时只有一个参数。如果 Kotlin 可以通过本身进行计算，那么可以省略这个唯一的参数，并通过 it 隐式声明它。

在 Kotlin 中，如果一个函数的最后一个参数是函数，并且传递一个 Lambda 表达式作为相应的参数，那么可以在括号外指定 Lambda 表达式中的值。示例代码如下：

```
ints.filter { it > 0 }
```

可以使用返回语句返回 Lambda 表达式的显式带有数据类型的值或者不显示数据类型的值。因此，下面两行代码是等价的。

```
ints.filter { val shouldFilter = it > 0 shouldFilter }
ints.filter { val shouldFilter = it > 0 return@filter shouldFilter }
```

【例 4.10】更改数组中的元素。

```
package jqiang.Lambdas
    fun main(args: Array<String>) {
        val a= arrayOf(1,2,3)
        val b=a.map { "第${it}个" }
        for (s in b) {
            println(s)
        }
    }
```

运行结果如下：

```
第 1 个
第 2 个
第 3 个
```

【例 4.11】对数组中的元素进行求和。

```
package jqiang.Lambdas
fun main(args: Array<String>) {
    val a= arrayOf(1,2,3,4,5)
    var d = a.forEach {
        c += it
```

```
    }
    println(c)
}
```

运行结果如下：

```
15
```

在大多数情况下，这是不必要的，因为返回类型可以自动推断出来。然而，如果确实需要携带有返回类型的值，则可以使用另一种语法形式的匿名函数。示例代码如下：

```
fun(x: Int, y: Int): Int = x + y
```

匿名函数看起来非常像一个常规函数的声明，除其名称省略外。它的函数体可以是表达式（如上所示），也可以是代码块。示例代码如下：

```
fun(x: Int, y: Int): Int { return x + y }
```

匿名函数的返回类型推断机制与正常函数一样，对于具有表达式函数体的匿名函数将自动推断返回类型，而具有代码块函数体的匿名函数的返回类型必须显式指定（或者假定为 Unit）。

注意：匿名函数的参数总是在括号内传递的，允许将函数留在括号外传递仅适用于 Lambda 表达式。不带标签的 return 语句总是在使用 fun 关键字声明的函数中返回，这意味着 Lambda 表达式中的 return 将从包含它的函数返回，而匿名函数中的 return 将从匿名函数自身返回。

4.9　协程

一些 API 启动长时间运行的操作（如网络 I/O、文件 I/O、CPU、GPU、密集型任务等），并要求调用者阻塞直到它们完成。协程提供了一种方法避免阻塞线程，并通过更廉价、更可控的操作来替代线程阻塞，如协程挂起。

协程通过将复杂性放入库来简化异步编程。程序的逻辑可以在协程中顺序地表达，而底层库会为我们解决其异步性。该库可以将用户代码的相关部分包装为回调、订阅相关事件，在不同线程（甚至不同机器）上调度执行，而代码则保持如同顺序执行一样简单。

使用 Kotlin 协程可以将在其他语言中可用的一些异步机制实现为库，包括源于 C#和 ECMAScript 的 async/await、源于 Go 的管道和 select，以及源于 C#和 Python 的生成器/yield。

4.9.1 阻塞 VS 挂起

基本上，协程计算可以被挂起而无须阻塞线程。阻塞线程的代价通常是昂贵的，尤其是在高负载时，因为只有相对少量的线程可用，因此阻塞其中一个线程会导致一些重要的任务被延迟执行。

另外，协程挂起几乎是无代价的，不需要上下文切换或者系统进行任何干预。最重要的是，挂起在很大程度上可以由用户库控制：作为库的作者，我们可以决定挂起时发生什么事情并根据需求优化/记录日志/截获。

协程不能在随机的指令中挂起，而只能在所谓的挂起点挂起，这会调用特别标识的函数。

当调用使用特殊的 suspend 修饰符标识的函数时，会发生挂起。代码如下：

```
suspend fun doSomething(foo: Foo): Bar { … }
```

这样的函数被称为挂起函数，因为调用它们可能会挂起协程（如果相关调用的结果可用，库可以决定继续进行而不挂起）。挂起函数能够以与普通函数相同的方式获取参数和返回值，但它们只能从协程和其他挂起函数中调用。

要启动协程，必须至少有一个挂起函数，它通常是匿名的（即它是一个挂起 Lambda 表达式）。让我们来看一个例子，一个简化的 async()函数（来自 kotlinx.coroutines 库），代码如下：

```
fun <T> async(block: suspend () -> T)
```

这里的 async()是一个普通函数（不是挂起函数），但是它的 block 参数具有带 suspend 修饰符的函数类型：suspend () -> T。

将 Lambda 表达式传递给 async()时，它是挂起 Lambda 表达式，可以从中调用挂起函数。示例代码如下：

```
async { doSomething(foo) … }
```

继续该类比，await()是一个挂起函数（因此可以在一个 async {}块中调用），该函数挂起一个协程，直到一些计算完成并返回结果。示例代码如下：

```
async { … val result = computation.await() …}
```

挂起函数 await()和 doSomething()不能在像 main()这样的普通函数中调用。示例代码如下：

```
interface Base { suspend fun foo() }
class Derived: Base { override suspend fun foo() { … } }
```

4.9.2　协程的内部机制

下面给出关于协程如何工作的完整解释，但是粗略地认识到发生了什么事情是相当重要的。

协程完全通过编译技术来实现（不需要来自 VM 或 OS 端的支持），挂起通过代码生效。基本上，每个挂起函数（优化可能适用，但不在这里讨论）都转换为状态机，其中的状态对应于挂起调用。刚好在挂起前，下一状态与相关局部变量等一起存储在编译器生成的类的字段中。在恢复该协程时，恢复局部变量并且状态机从刚好挂起之后的状态进行。

挂起的协程可以保持其挂起状态并作为局部变量的对象来存储和传递。这种对象的类型是延续性的，而这里描述的整个代码转换对应于经典的延续性传递风格（Continuation-passing style）。因此，挂起函数有一个 Continuation 的类型的额外参数作为高级选项。

4.10　本章小结

本章主要讲解了 Kotlin 中函数的定义及调用、高阶函数、内联函数、Lambda 表达式的使用以及协程。希望通过本章内容的学习，读者可以熟练使用函数，使自己的程序具有可读性。

第5章

Kotlin 面向对象

面向对象的思想涉及软件开发的各个方面，如面向对象的分析（Object Oriented Analysis，OOA）、面向对象的设计（Object Oriented Design，OOD）和面向对象的编程（Object Oriented Programming，OOP）。

面向对象的分析根据抽象关键的问题域来分解系统。面向对象的设计是一种提供符号设计系统的面向对象的实现过程，它用非常接近于实际领域术语的方法把系统构造成“现实世界”的对象。面向对象的程序设计可以看作一种在程序中包含各种独立而又相互调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是对电脑下达的一系列指令。面向对象程序设计中的每一个对象都应该能够接收数据、处理数据并将数据传达给其他对象，因此它们都可以被看作一个小型的“机器”，即对象。

面向对象是编程语言中最关键也是最重要的，希望读者认真学习，将所学应用到实践中。

本章主要内容：

- 面向对象的基本概念
- Kotlin 中的面向对象
- 类的定义及实例化
- 声明类的成员
- 面向对象的三大特性
- 数据类的创建和实现
- 枚举类的创建和实现
- 密封类的创建和实现
- 抽象类的实现

- 接口的使用
- 泛型
- 异常
- 创建包和导入包

5.1　面向对象的基本概念

这里所说的面向对象，是指面向对象编程(OOP)。面向对象包括三个部分：面向对象分析（OOA）、面向对象设计（OOD）和面向对象编程（OOP）。类和对象是面向对象的重要概念。

5.1.1　类

万事万物都具有其自身的属性和方法，通过属性和方法可以表现出不同的性质，如人具有身高、性别和肤色等属性，还有吃饭、运动和走路等动作，这些动作可以理解为人所具有的功能。假如把人看成程序的一个类，那么身高就可以看成类的一个属性，运动可以看成类的方法。也就是说，类是由属性和方法组成的。类是面向对象的核心和基础，可以将相似的对象封装在一个类中，以方便进行有效的管理，比如创建一个运动员类，包含姓名、身高、体重、年龄和性别等属性，方法可以定义为踢足球、打篮球等。

5.1.2　对象

对象是类的实例化，创建一个对象表示实例化一个类，因此“类的实例化”和“对象”具有相同的含义。只有创建对象，才可以使用类的属性和方法。

5.1.3　面向对象的三大特性

面向对象的三大特性是：封装、继承和多态。

1．封装

封装也可以称之为“信息隐藏”，是指将类的使用和实现分开，只保留有限

的接口提供给外部使用。对于开发人员来说，只要能够使用该类就行，而不关心类是怎么实现的。

封装只留给开发者一个访问对象的接口，使开发者不能直接访问内部信息，这就保证了类中数据的安全性。面向对象的封装示意图如图 5-1 所示。

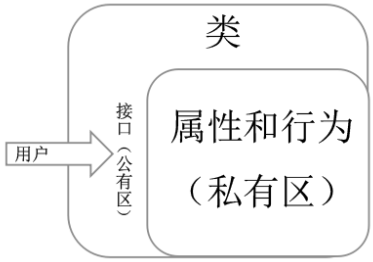


图 5-1

2．继承

一个类（子类）可以使用另外一个类（父类）的属性和方法，这种特性就是继承，反之则称作“派生”。派生类（子类）自动继承一个或者多个基类（父类）中的属性和方法，并且可以重写或者添加新的属性和方法。这样做的目的是可以减少类和对象的创建，提高代码的可复用性。

比如公司的雇员（Employe）派生为销售员和经理两个类，销售经理继承了销售员和经理两个类的共同特性，如图 5-2 所示，其中箭头所指表示被继承的父类。

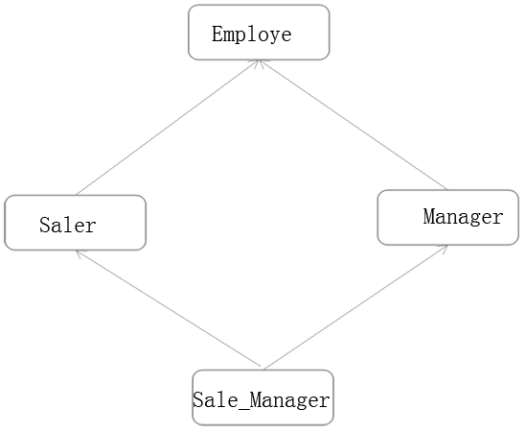


图 5-2

从图 5-2 可以看出，一个父类可以被多个子类继承（称为单继承），一个子类也可以继承自多个父类（称为多继承）。Saler 类与 Employe 类之间就是单继承的关系，因为 Saler 类只有一个直接父类 Employe；同样，Manager 类与 Employe 类之间也是单继承的关系。而 Sale_Manager 类因为同时继承自父类 Saler 和 Manager，所以它们之间是多继承的关系。

Kotlin 和 Java 如出一辙，所以 Kotlin 只支持单继承方式，这使得 Kotlin 的继承方式比 C++等面向对象语言要简单得多。虽然 Kotlin 不支持多继承方式，但是并不意味着 Kotlin 不能实现多继承，可以通过接口（Interface)实现多继承。运用继承机制可以提高代码的可复用性，提高软件的开发效率，降低程序产生错误的可能性，并为程序的修改、扩展提供便利。

3．多态

多态就是指同一个操作（方法）作用于不同的对象时，可以有不同的解释，产生不同的执行结果。

比如在实际生活中，H_2O 在不同的温度下可能表现为固态、液态或者气态。这就是多态性的体现。而在编程中，简单地讲，就是“类的不同对象可以对同一个消息做出不同的响应”。

5.2　类与对象

对象：是人们要进行研究的任何事物，它不仅能表示具体的事物，还能表示抽象的规则、计划或事件。对象具有状态，对象用数据值来描述其状态；对象还有操作，用于改变对象的状态，其中数据值就是对象的成员属性，操作以及改变状态就是对象的成员方法。对象实现了数据和操作的结合，使数据和操作封装于对象的统一体中。

类：对具有相同特性（数据元素）和行为（功能）的对象的抽象。因此，对象的抽象是类，类的具体化就是对象，也可以说类的实例是对象。类实际上就是一种数据类型。类具有属性，它是对象的状态的抽象，用数据结构来描述类的属性。类具有操作，它是对象的行为的抽象，用操作名和实现该操作的方法来描述。

类和对象是面向对象编程中最基本的两个概念，类是对象的模板，对象是类的具体实现。如图 5-3 所示，通过观察对象的属性和方法来了解对象，对象的属

性用于描述对象的状态，对象的方法用于描述对象的功能。

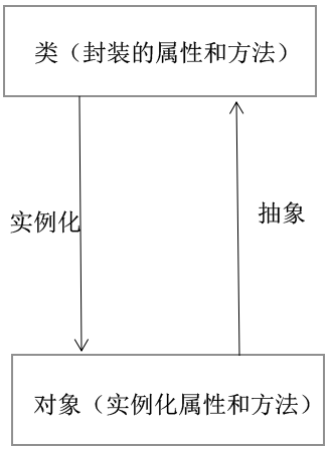


图 5-3

“实例化”是将类的属性设定为确定值的过程，即从“一般”到“具体”；“抽象”是从特定的实例中抽取出共同的性质，形成一般化概念的过程，即从“具体”到“一般”。

假如学生是一个类，教室里所有的同学是对象，他们都具有学生的基本特征，但是每个同学的特征又是不一样的，比如身高、体重、学习成绩等不一样，当这些特征确定以后，一个完整的学生类就确定下来，此时学生就是对象。

如果对所有同学的共同特征进行总结，则会得到如下信息：凡是学生都有性别、年龄、班级、选修专业等属性特征，而且都有学习成绩、说话方式等行为特征。把这些共同特征集成到一个模块中就形成了类，这就是类抽象的过程。

在面向对象编程中，“类”是对具有相同类型的成员属性和成员方法的封装。“对象”是类的具体存在。类是作为对象的蓝图而存在的，所有的对象都依据相应的类来生成。

5.2.1 类的定义

类是对某个对象的定义，其中包含了对象中的成员属性和成员方法。实际上类本身并不是对象，因为它不存在于内存中。当引用类的代码运行时，就在内存中创建了类的一个新实例即对象。虽然只有一个类，但是在内存中可以通过这个类创建多个具有相同类型的对象。

可以把类看作“理论上”的对象，也就是说，类为对象提供蓝图，通过这个蓝图可以创建任何数量的对象，但是类在内存中并不存在，类实例化后才可以使用该类的成员属性和成员方法。例如学生类，既可以是一个类，也可以是一个对象。其中学生有不同的年龄、不同的喜好，这就是类中的成员属性和成员方法。如果要使用该类的成员属性和成员方法，则应该先进行实例化。

跟很多编程语言一样，Kotlin 也是通过 class 关键字来声明类的，其中类由构造函数和初始化模块、函数、属性、嵌套类和内部类、对象声明 5 部分组成。

类是用户自定义的类型，如果在程序中要使用类，则必须提前声明类，或者使用已存在的类（别人写好的类、标准库中的类等）。定义一个简单的 PlayerObject 类，一般格式如下：

```
package jqiang.oop
class PlayerObject{}
```

使用大括号括起来的部分是类的全部内容。PlayerObject 是一个简单的类，仅有类的骨架，什么功能都没有实现，但是不影响它的存在。如果一个类中没有大括号里面的形参和函数部分，那么大括号可以省略不写。代码如下：

```
package jqiang.oop
class PlayerObject
```

5.2.2　成员属性

类的变量也称为成员变量，成员变量用来保存数据信息，或者与成员方法进行交互实现某个功能。在 Kotlin 中，根据定义的成员属性的位置不同，大致可以分为普通类型的成员属性和组合型成员属性，其中普通类型的成员属性和定义变量的方式有点类似。

类的成员变量在使用前必须先声明，除声明变量的数据类型外，还要声明变量的访问属性和存储方式。访问修饰符有：public、protected、private、internal，也可以缺省（就是不带访问修饰符）；存储类型修饰符有：open、override 和 const。

成员变量根据其在内存中的存储方式和作用范围可以分为：类变量和实例变量，普通变量也被称之为实例变量。使用 open 修饰的成员变量，表示该成员变量允许被继承；否则，该成员变量是不允许被继承的。类的成员变量在该类的任何地方都可以使用，但是如果外部需要调用该成员变量，则需要先进行实例化。

定义成员变量，格式如下：

```
关键词 成员变量名[数据类型] = 默认值
```

实例化对象之后，访问成员属性或成员方法的方式与访问成员变量一样，都需要使用点方法进行访问。格式如下：

```
对象名.成员变量
```

使用 const 修饰的成员变量称为常成员变量，在 Kotlin 中常成员变量习惯使用大写的标识符，例如：CONST val double = 3.1415926。

需要强调的是，在一个类中定义成员变量，需要在成员方法之外声明定义，其作用域是当前整个类。如果是在成员方法内部定义变量的，那么这个变量只能在当前方法中进行调用，其作用域是当前类中的成员方法。

【例 5.1】定义一个 PlayerObject 类，并添加普通属性，如姓名、体重、身高，以及组合属性，如健康程度（体重/身高的平方得到的值，小于 18.5 表示偏瘦，在 18.5 和 23.9 之间表示正常，在 24.0 和 27.9 之间表示过重，大于或等于 28.0 表示肥胖）。

```
package jqiang.oop
class PlayerObject{
    var name:String="Kotlin"
        set
    var weight=0.0
        set
    var height=0.0
        set
    var BIM:String=""
        get(){
        protected var b=this.weight/Math.pow(this.height, 2.0)
        if(b<=18.5){
            return"偏瘦"
        }elseif(bin18.5..23.9){
                return"正常"
            }elseif(bin24.0..27.9){
            return"过重"
        }else{
            return"肥胖"
        }
        }
}
fun main(args:Array<String>){
    var Player=PlayerObject()
    Player.weight=80.5//性别
```

```
        println("您好${Player.name}，性别：${Player.sex}，您的身高是
${Player.height}米，体重是${Player.weight}公斤，您的身体状况为
${Player.BIM}")
    }
```

protected 的意思是受保护的，主要用来进行访问修饰，即凡是被 protected 修饰的都不能被外部访问。还有其他一些访问修饰符，如 public、private 和 internal。

main 函数又称主函数，它是程序执行的起点。main 是相对来说的，如同音学理论中的主调与泛音，泛音即程序中除 main 之外的其他函数，迎合人们的思考方式而生成的非必定的模式，有主有次，执行起来条清缕析，既可将程序模块化，又实现了一个闭合的整体。

细心的读者会发现在实例化对象时，没有使用 new 关键字，这就是 Kotlin 简单、快捷的体现，不需要使用 new 关键字就可以将对象实例化。

5.2.3　成员方法

类的函数被称为成员方法。函数和成员方法唯一的区别就是，函数实现的是某个独立的功能；而成员方法实现的是类的行为，是类的一部分。

成员方法主要用于实现特定的操作，如进行数据处理、显示等，它决定了类的实现功能，在一个类中方法名不能相同。

定义成员方法需要使用 fun 关键字进行声明。除声明方法的返回值之外，还要声明成员方法的访问属性和存储方式。访问修饰符有：public、protected、private、internal，也可以缺省（就是不带访问修饰符）；存储类型修饰符有：open、override。

如果一个类中的成员方法没有加任何修饰符，那么该方法是 final 类型的，这种方法默认是不允许被子类继承的。如果允许被子类继承，那么需要在该类的方法前加上 open 修饰符。

定义成员方法的一般格式如下：

```
fun 方法名（属性列表）{方法体}
```

访问成员方法的格式如下：

```
对象名.方法名
```

成员方法按照返回值类型分为两类：无返回值类型的方法和有返回值类型的方法。对于有返回值类型的方法，其返回值类型可以是 Kotlin 中的任何数据类型；对于无返回值类型的方法，就是直接通过 Kotlin 中的 println 函数进行打印，无须

使用 return 语句。对于有返回值类型的方法，可以在方法后面添加返回值的数据类型，并且包含 return 语句，return 后面就是返回的结果，结果的数据类型即为函数返回的数据类型。

【例 5.2】创建一个运动员类，命名为 PlayerObject，并添加打篮球的成员方法 playBasketball()。

```
package jqiang.oop
    class PlayerObject{
        var name:String=""
            set
        fun playBasketball(){
            println("${this.name}可以打篮球")
        }
    }
```

在 PlayerObject 类中声明一个成员变量 name 和一个成员方法 playBasketball()，该方法直接打印出谁在打篮球。如果需要在其他地方通过 print 函数打印该函数的返回值，则只需要将该成员方法的值通过 return 返回即可。

5.2.4 对象实例化

类是一种数据类型，对象相当于这种数据类型的变量，它在使用前要先定义或者声明。例如，对于上面定义的 PlayerObject 类，将对象赋值给变量 Player，使用语句 var Player=PlayerObject()。

声明对象之后，并没有创建该对象。首先要把对象赋值给变量，然后通过该变量访问成员变量和成员方法。代码如下：

```
var Player=PlayerObject()
Player.weight=58.0//体重
```

在 5.2.1 节和 5.2.2 节讲过，成员变量用于存储数据信息，成员方法将成员属性重新进行业务处理后返回所需要的值。在使用该类的成员属性和成员方法之前，应对该对象进行实例化，然后通过点方法进行访问、赋值等操作。

【例 5.3】以 PlayerObject 类为例，实例化对象之后调用成员变量和成员方法 playBasketball()。

```
fun main(args:Array<String>){
    var Player=PlayerObject()
    Player.name="jqiang"
```

```
        Player.weight=48.0//体重
        Player.height=1.60//身高
        println("您好${Player.name}，您的身高是${Player.height}米，体重是
${Player.weight}公斤，您的身体状况为${Player.BIM}")
        Player.playBasketball()
    }
```

运行结果如下：

```
您好 jqiang，您的身高是 1.6 米，体重是 48.0 公斤，您的身体状况为正常
jqiang 可以打篮球
```

在上面例子中没有使用 new 关键字；访问变量和成员方法使用的是点。

在 Kotlin 中，实例化一个对象不需要使用 new 关键字，只需要将该对象赋值给变量 Player 即可。

使用点既可以给变量赋值，也可以访问变量，还可以访问成员方法 playBasketball()。

5.2.5　构造函数

构造函数是一种特殊的方法，主要用来在创建对象时初始化对象，即为对象成员变量赋初始值。构造函数总是与 new 关键字一起使用在创建对象的语句中。在一个类中可以有一个主构造函数和多个次构造函数，也可以没有次构造函数

主构造函数不能包含任何代码，初始化代码可以放在以 init 关键字作为前缀的初始化模块中，在主函数声明后可以当作全局变量使用。

```
clss Person(val name:String){
    init{
 var leng=name.length
 }
    val leng=name.length
    fun le(){
 val leng=name.length
 }
}
```

如果有一个主构造函数，那么每个次构造函数都需要直接或者间接委托给主构造函数，使用 this 关键字。

```
clss Person(val name:String){
    constructor(){}
```

```
    constructor(name:String):this(){}
    constructor(name:String ,age:Int):this(this){}
}
```

当类被实例化之后，随着该类的初始化，类中的成员属性和成员方法也将被初始化。

在 PlayerObject 类中添加一些成员变量，示例代码如下：

```
package jqiang.oop
class PlayerObject{
    var name:String="Kotlin"
        set
    var weight=0.0
        set
    var height=0.0
        set
    var BIM:String=""
        get(){
            var b=this.weight/Math.pow(this.height, 2.0)
            if(b<=18.5){
                return"偏瘦"
            }elseif(bin18.5..23.9){
                return"正常"
            }elseif(bin24.0..27.9){
                return"过重"
            }else{
                return"肥胖"
            }
        }
    var sex:Boolean=true
        set
}
```

实例化 PlayerObject 类，可以通过点方法对成员变量进行赋值和取值。

```
var Player=PlayerObject()
Player.name="jqiang"
Player.weight=48.0//体重
Player.height=1.60//身高
Player.sex=true//性别
```

可以看到，如果赋值比较多，那么写起来非常麻烦，为此，Kotlin 还定义了一个类，称之为主构造函数。

主构造函数的定义格式一般如下：

```
class PlayerObject constructor(firstName:String){}
```

【例 5.4】重写 PlayerObject 类和 playBasketball()方法。

```
class PlayerObject constructor(var name:String, var weight:Double,
var height:Double){
    var sex:String=""
        set
    valBIM:String
    get(){
        var b=this.weight/Math.pow(this.height, 2.0)
        if(b<=18.5){
            return"偏瘦"
        }elseif(bin18.5..23.9){
            return"正常"
        }elseif(bin24.0..27.9){
            return"过重"
        }else{
            return"肥胖"
        }
    }
    funplayBasketball(){
        println("我可以打篮球")
    }
}
fun main(args:Array<String>){
    var Player=PlayerObject("jqiang", 48.0, 1.60)
    Player.sex="男"//性别
    println("您好${Player.name}，性别：${Player.sex}，您的身高是
${Player.height}米，体重是${Player.weight}公斤，您的身体状况为
${Player.BIM}")
    Player.playBasketball()
}
```

运行结果如下：

```
您好 jqiang，性别：男，您的身高是 1.6 米，体重是 48.0 公斤，您的身体状况为正常
我可以打篮球
```

构造函数在初始化对象时使用，如果类中没有任何注解或可见修饰符，则可以省略 constructor 关键字。代码如下：

```
class PlayerObject(var name:String , var weight:Double , var
```

```
height:Double){
    }
```

在类中除可以声明主构造函数之外，也可以通过 constructor 关键字声明次构函数，代码如下：

```
    class Person{
    constructor(parent:Person){
        parent.children.add(this)
        }
    }
```

如果类有一个主构造函数，每一个次构造函数都需要委托给主构造函数，则可以直接委托或者通过其他的次构造函数间接委托。委托给同一个类中的另一个构造函数，使用 this 关键字。代码如下：

```
    class Person(val name:String){
        constructor(name:String, parent:Person):this(name){
        parent.children.add(this)
        }
    }
```

对实例属性的设置有两种方式：一种是实例化对象之后，通过点方法访问成员属性和成员方法；另一种就是直接在实例化对象时调用主构造函数，一次完成实例的创建和成员变量的赋值。这就称为“初始化”。

比较这两种方式，就可以很容易地知道主构造函数的使用时机和作用：主构造函数主要在实例化对象时对成员变量进行赋值操作。

5.2.6 继承和多态的实现

面向对象编程语言的一个主要功能就是“继承”。通过继承，可以使用现有类的所有功能，并且在不需要重新编写现有类的情况下对这些功能进行扩展。

1. 继承

继承是面向对象最显著的一个特性。继承是指从已有的类派生出新类，新类能使用已有的类的数据属性和行为，并能扩展新的能力。Kotlin 继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。这种技术使得复用以前的代码非常容易，能够大大缩短开发周期，降低开发费用。比如可以先定义一个类

“车”，“车”有车体大小、颜色、方向盘、轮胎等属性，由“车”这个类可以派生出“轿车”和“卡车”两个类，为“轿车”添加一个小后备箱，而为“卡车”添加一个大货箱。

通过继承创建的新类称为“子类”或“派生类，被继承的类称为“基类”“父类”或“超类”。继承的过程，就是从一般到特殊的过程。

继承的三种方式：实现继承、接口继承和可视继承。

- 实现继承是指使用基类的属性和方法；
- 接口继承是指仅使用属性和方法的名称；
- 可视继承是指子窗体（类）使用基窗体（类）的外观和实现代码。

抽象类仅定义由子类创建的一般属性和方法，在创建抽象类时，请使用关键字 Interface 而不是 Class。

在 Kotlin 中所有的类都有一个共同的超类 Any，没有超类型声明的类是默认超类

```
class Examole
```

Any 不是 java.lang.Object；除了 equals()、hashCode()和 toString()，它没有任何成员。

下面我们看在 Kotlin 中如何实现继承类。

```
open class Base(b:Int)
class SonClass(b:Int):Base(b)
```

类中的 open 关键字和 Java 中的 final 关键字的作用相反，一个类如果想要被其他类继承，就需要对这个类的属性使用 open 关键字修饰。同理，如果需要重写该类的成员属性或者成员方法，也需要使用 open 关键字修饰。在默认情况下，Kotlin 中所有的类、所有的成员属性和成员方法都是使用 final 关键字修饰的。

【例 5.5】使用 SportObject 类生成一个子类 BeatBasketBall，并且输出信息。

```
package jqiang.oop
open class SportObject(var name:String, var weight:Double, var
sex:String){
    open fun showMe(){
        if(this.weight>185){
            println("${this.name}，符合打篮球标准")
    }else{
            println("${this.name}，不符合打篮球标准")
        }
    }
```

```
    }
    class  BeatBasketBall(name:String ,  weight:Double ,  public  val
height:Double, sex:String):SportObject(name, weight, sex){
        override fun showMe(){
            if(this.height<85){
                println("${this.name}, 符合举重标准")
            }else{
                println("${this.name}, 不符合举重标准")
            }
        }
    }
    fun main(args:Array<String>){
        var Sport=SportObject("jqiang", 100.0, "男")
        Sport.showMe()
        var BeatBasketBall=BeatBasketBall("jqiang", 120.0, 58.0, "男")
        BeatBasketBall.showMe()
    }
```

运行结果如下：

```
jqiang, 不符合打篮球标准
jqiang, 符合举重标准
```

在继承过程中一定要注意，父类需要使用 open 关键字修饰之后才能被继承；父类的成员属性和成员方法也需要被 open 关键字修饰之后才可以进行覆盖重写。接口、接口方法、抽象类默认被 open 关键字修饰，覆盖父类的接口成员需要使用 override 关键字。

2．多态

多态可以理解为一个类中的成员方法，不同的人有不同的特长，有些人特长是唱歌，有些人特长是跳舞，有些人特长是写，还有些人什么都不会。虽然是同一个方法，却产生不同的形态，这就是多态。

多态允许不同类的对象对同一消息做出响应，即同一消息可以根据发送对象的不同而采用多种不同的行为方式。

多态通常表现为覆盖和重载。

覆盖：指子类重新定义父类的函数。在开发中父类的方法往往不能满足需求，所以在创建子类时可以对父类的方法进行重写（重新实现），但是调用方法名是一样的，通过调用不同的类返回不同的值。就像例 5.5 中的 showMe()方法，返回的结果不同。

覆盖（重写）的使用规则如下：

- 重写方法的参数列表必须完全与被重写的方法相同；否则不能称其为重写，而是重载。
- 重写方法一定要被 open 关键字修饰。
- 重写方法的返回值必须和被重写方法的返回值一致。

重载：一般用于在一个类中实现若干重载的方法，这些方法的名称相同而参数形式不同。

重载的使用规则如下：

- 在重载时只能通过相同的方法名、不同的参数形式来实现。不同的参数形式可以是不同的参数类型、不同的参数个数、不同的参数顺序（参数类型必须不一样）。
- 不能通过访问权限、返回类型、抛出的异常进行重载。
- 假设有多个同名的函数，它们的参数个数和参数类型不同，在调用时，可以根据传入的参数个数或参数类型自动调用对应的函数。

【例 5.6】实现一个简单的重载，根据传入的参数来判断调用的方法。

```
package jqiang.oop
class Burders(var a:Boolean){
    init{
        if(this.a==true){
            this.show()
        }else{
            this.say()
        }
    }
    fun show(){
        println("show 方法")
    }
    fun say(){
        println("say 方法")
    }
}
fun main(args:Array<String>){
    var b=Burders(false)
}
```

在继承关系中，父类通用，子类具体。父类具有一般的特征和行为，而子类除了具有父类的特征和行为，还具有一些自己的特征和行为。

当两个类具有相同的特征（属性）和行为（方法）时，可以将相同部分抽取出来放到一个类中，将这个类作为父类，这两个类继承这个父类。

子类自动继承父类的属性和方法，在子类中可以定义特定的属性和方法。子类可以重新定义父类的属性、重写父类的方法，以获得与父类不同的功能。

如果在子类中定义的一个方法，其名称、返回类型及参数列表正好与父类中某个方法的名称、返回类型及参数列表相匹配，那么就可以说子类的方法重写了父类的方法。

多态与继承、方法重写密切相关，在方法中接受父类类型作为参数，在方法实现中调用父类类型的各种方法。当把子类作为参数传递给这个方法时，Java 虚拟机会根据实际创建的对象类型，调用子类中相应的方法（存在方法重写时）。

5.2.7 封装

面向对象编程的特点之一是封装，也可以称之为“数据隐藏”，就是把过程和数据包围起来，对数据的访问只能通过已定义的接口。

面向对象计算始于这个基本概念，即现实世界可以被描绘成一系列完全自治、封装的对象，这些对象通过一个受保护的接口访问其他对象。

在 Kotlin 中通过 private、protected、internal 和 public 修饰符实现封装。封装把对象的所有组成部分组合在一起，定义程序如何引用对象的数据，实际上是使用方法将类的数据隐藏起来，控制用户对类的修改和访问数据的程度。适当的封装可以让代码更容易理解和维护，也提高了代码的安全性。

封装的好处如下：

（1）良好的封装可以减少耦合。

（2）用户只能通过指定的方法访问数据，可以方便地加入控制逻辑，限制对属性的不合理操作。

（3）便于修改，增强了代码的可维护性。

（4）可以进行数据检查。

【例 5.7】封装一个类，该类完成两个数的加、减、乘、除，并针对 num1、num2 和 operator 三个变量进行不同的权限赋予，然后输出运算结果。

```
package jqiang.oop
class CaculatorNum(protected var num1:Int, private var num2:Int){
    public var operator:Char='+'// +、-、*、/
    fun getnum1(){
```

```
        println("${num1}")
    }
    internal fun getnum2(){
        println("${num2}")
    }
    fun caculatNum(){
        when(this.operator){
            '+'->{
                println("两个数相加：${this.num1+this.num2}")
            }
            '-'->{
                println("两个数相减：${this.num1-this.num2}")
            }
            '*'->{
                println("两个数相乘：${this.num1*this.num2}")
            }
            '/'->{
                println("两个数相除：${this.num1/this.num2}")
            }
        }
    }
}
fun main(args:Array<String>){
    var CaculatorNum=CaculatorNum(100, 20)
    CaculatorNum.operator='-'
    //CaculatorNum.num1;// 错误：受保护的变量无法直接访问和赋值
    //CaculatorNum.num2;// 错误：私有变量无法直接访问和赋值
    CaculatorNum.getnum1();
    CaculatorNum.getnum2();
    CaculatorNum.caculatNum()
}
```

运行结果如下：

```
100
20
两个数相减：80
```

为了保护数据的完整性和安全性，所以在封装的过程中通过 public、protected、private 和 internal 修饰符对成员变量和方法进行保护。

对于 public 修饰符，表示被 public 修饰的数据成员、成员函数对所有用户开放，所有用户都可以直接进行调用。

对于 protected 修饰符，表示被 protected 修饰的成员属性和成员方法可以被当

前类或者子类随意访问，但是如果被其他类访问，就会变成私有类型的成员属性和成员方法。

对于 private 修饰符，表示被 private 修饰的访问权限仅限于类的内部，是一种封装的体现。例如，大多数成员变量都是使用 private 修饰的，它们不希望被任何外部的类访问。

对于 internal 修饰符，表示被 internal 修饰之后，虽然也可以像被 public 修饰一样随意使用，但是仅仅限于在当前模块下使用；如果其他模块需要使用，就会变成私有类型。

【例 5.8】创建一个父类 Outer，并创建两个子类继承自这个父类，查看 public、protected、private 和 internal 修饰符

```
package jqiang.oop
open class Outer{
    privatevar  a=1// 私有的，当前类可见，最小的可见性
    protectedvar b=2// 受保护的，仅子类可见
    internalvar c=3// 内部的，当前模块可见
    open publicvar d=4// 公开的，完全可见
}
class subclass:Outer(){
    funtest(){
        println(super.b)
        println(super.c)
        println(super.d)
    }
}
class Unrelated(publicvalo:Outer){
    funtest(){
        println(o.c)
        println(o.d)
    }
}
```

在 Kotlin 中，对象的数据封装特性彻底消除了在传统结构方法中数据与操作分离所带来的种种问题，提高了程序的可复用性和可维护性。

另外，对象的数据封装特性还可以把对象的私有数据和公共数据分离开，以保护私有数据，减少可能的模块间干扰，达到降低程序复杂性、提高可控性的目的。

5.3　Kotlin 对象高级应用

经过对 5.2 节内容的学习，相信读者对 Kotlin 的面向对象基础有了一定的了解和认识，接下来我们就一起来学习面向对象的高级应用。

5.3.1　this 关键字的使用

通过例 5.5 可以发现，子类不仅可以调用自己的变量和方法，也可以调用父类的变量和方法，而且还可以调用其他不相关的类成员。

在 5.2 节中，我们已经对如何调用成员方法有了一定的了解，就是对象名加上方法名，即：对象名.方法名()。但是在定义类时，根本不知道对象的名称是什么，这时候如果调用该类的方法，就使用伪变量 this。this 的意思是本身，所以 this 只可以在类的内部使用。

this 关键字也可以在构造函数中进行传递，比如 this(a,b)表示调用另一个构造函数。这里的 this 就是一种特殊语法，不是变量，没有类型。

【例 5.9】当类被实例化后，this 同时被实例化为该类的对象，这时候对类使用 javaClass 方法，将打印类名。

```
package jqiang.hight
class athis(){
    private var a:String="this is a"
    fun showclass(){
        println(this.javaClass)
    }
    fun seta(a:String): String {
        this.a=a
        return a
    }
    fun geta(): String {
        return this.a
    }
}
fun main(args:Array<String>){
    var athis=athis()
    athis.seta("this is test")
    print(athis.geta())
}
```

运行结果如下：

```
this is test
classjqiang.oop.athis
```

this 表示类实例的本身，在 athis 类中，this.javaClass 表示当前对象的类名，this.a 表示当前对象定义的变量，无论是以普通的方式还是以主构造函数的方式定义类，都可以使用 this 调出当前对象的属性或方法。

5.3.2 super 关键字的使用

super，与 this 类似，但是 super 既可以代表父类的成员变量，也可以代表父类的方法，还可以在一个类的成员内部使用，比如 super.play()。

super 的另一个作用是调用父类的 protected 函数。只有通过 super，我们才能操作父类的 protected 成员，别无他法。

【例 5.10】创建一个父类和一个子类，子类调用父类的成员属性和方法。

```
package jqiang.hight
open class superfather {
    var a=10
    fun message () { println("this is class superfather fun") }

}
class son:superfather(){
    fun show(){
        super.message()
        println(super<superfather>.a)
        println("this is class son fun")
    }
}
fun main(args: Array<String>) {
    val son =  son()
    son.show();
}
```

son 继承了 superfather 类，super 只能在子类方法中调用父类的成员变量和方法，如果 son 继承了多个类，则可以使用“super<继承的类名>.成员变量”或者“super<继承的类名>.方法”形式，这样就解决了当子类继承多个父类时，不知道该调用哪一个父类的成员属性或者方法的问题。

5.3.3　open 关键字的使用

open，中文意思是“开着的”“公开的”“公共的”。使用 open 修饰的类和方法才能被子类重新编写和覆盖。定义一个可以被继承的类，一般形式如下：

```
open class Preson(){}
```

这么定义之后，该类就是可以被继承的，它可能含有子类。如果没有使用 open 修饰，则该类不允许被继承。

【例 5.11】通过实例验证没有被 open 关键字修饰过的类是否可以被继承。

```
package jqiang.hight
class father
class son:father()
fun main(args:Array<String>){
}
```

运行结果如下：

```
Error:(3, 13)Kotlin:Thistypeisfinal, soitcannotbeinheritedfrom
```

在其他编程语言中，使用 final 关键字修饰的类才能被继承和覆盖，但是在 Kotlin 中却是截然相反的，类和成员变量默认就是 final 类型的，如果想要被继承或修改，就需要在类和成员变量前加 open 关键字。

5.3.4　嵌套类

在一个类中可以再创建一个类，这个类中的类就叫作嵌套类。

嵌套类不需要使用任何特殊的关键字修饰，直接在主类中创建一个类即可。一般格式如下：

```
class Person{class Language(){}}
```

内部类则略有不同，需要在类的前面加一个 inner 关键字来声明它是内部类，内部类只为主类服务。一般格式如下：

```
class Person{inner class Name(){}}
```

【例 5.12】创建一个主类 Person，该类有一个嵌套类 Language 和一个内部类 Names，在嵌套类中输出一个由数组构成的字符串，在内部类中有一个 changeName()方法，用于修改 Person 中的变量 Name 的值。

```
package jqiang.hight
class Person{
    private var Name="Kotlin"
    class Language{
        var list=arrayListOf("中文", "英语", "日语")
        var str=list.joinToString()
    }
    inner class Names{
        fun changeName(newName:String){
            Name=newName
            println("您可以看到新的名字是${Name}")
        }
    }
}
fun main(args:Array<String>){
    println(Person.Language().str)
    Person().Names().changeName("jqiang")
}
```

运行结果如下：

```
中文，英语，日语
您可以看到新的名字是 jqiang
```

在使用嵌套类时，首先应加一个主类名称前缀，然后实例化嵌套类，最后调用嵌套类的成员属性和方法。

在使用内部类时，首先必须要实例化主类，然后再实例化内部类，最后调用内部类的成员属性和方法。

5.3.5 数据类

在开发程序时，我们经常需要对数据进行增、删、查、改操作。例如，我们可以打开新闻网站读取最新的新闻；可以对自己的 QQ 空间中的日志文章进行增加、修改、删除等，还可以对 QQ 好友开放查看文章权限……这些都是数据，因此在 Kotlin 中提供了一个数据类。

在 Kotlin 中定义数据类的一般格式如下：

```
data class 类名(属性列表)
```

一般数据类中不应该有方法，所以将变量放在属性列表中，把大括号去掉，这样才是一个数据类。

【例 5.13】创建一个文章数据类，该数据类中存储有 id 号、文章标题（name）和文章描述（dec）三个变量。

```
package jqiang.hight
data class article(var id:Int, var name:String, var dec:String)
fun main(args:Array<String>){
    var article=article(1, "Kotlin", "全栈编程语言")
    println(article.toString())// 序列化
    var newName=article.copy(id=2, name="Java", dec="面向对象编程语言")
    println(newName.toString())// 更改属性
    var(id, name, dec)=article
    println("${id}, ${name}, ${dec}")// 解构
println("${article.component1()}  ,  ${article.component2()}  ,
${article.component3()}")// component 方法
}
```

运行结果如下：

```
article(id=2, name=Java, dec=面向对象编程语言)
1, Kotlin, 全栈编程语言
1, Kotlin, 全栈编程语言
```

总结：

（1）编辑器会自动到主构造函数中声明所有的属性，这时候我们需要使用方法输出对应的值。

（2）将数据类型转换成字符串，使用 toString()函数。

（3）可以在实例化之后访问指定元素，这样定义变量：var(id，name，dec)=article。

（4）复制或者修改其中的变量，使用 copy()。

5.3.6　枚举类

枚举类型在 C#、C++、Java、VB 等编程语言中是一种基本数据类型，而不是构造数据类型；而在 C 语言中是一种构造数据类型。枚举类型用于声明一组命名的常数，当一个变量有多个可能的取值时，可以将它定义为枚举类型。

枚举可以根据 Integer、Long、Short 或 Byte 数据类型来创建一个新变量，这个变量能被设置作为已经定义的一组变量中的一个，可以有效地防止用户提供无效值。该变量可使代码更加清晰，因为它可以描述特定的值，比如颜色（赤、橙、黄、绿、青、蓝、紫）、方向（东、南、西、北）等有限的数据，其中每一个数

据称为枚举常量，各个枚举常量之间使用逗号隔开。

枚举变量的声明与赋值：定义枚举类型后,可以声明该枚举类型的变量，然后使用该变量存储枚举元素的数值。

枚举变量：用枚举类型定义的变量。变量和参数都可以定义为枚举类型，过程的返回值也可以是枚举类型的。

在 Kotlin 中，声明一个枚举类需要使用 enum 关键字。一般格式如下：

```
enum class 类名{常量列表}
```

枚举成员是该枚举类型的命名常量。任意两个枚举成员不能具有相同的名称。每个枚举成员均具有相关联的常量值，此值的类型就是枚举的基础类型，每个枚举成员的常量值必须在该枚举的基础类型的范围之内。

【例 5.14】创建两个枚举类，分别是 rainbow（彩虹的颜色）和 corour（颜色的 RGB 值）。

```
package jqiang.hight
enum class rainbow{
    赤，橙，黄，绿，青，蓝，紫
}
enum class corour(valrgb:Int){
    RED(0xFF0000), GREEN(0x00FF00), BLUE(0x0000FF)
}
enum class anonys{
    WATTING{
        override fun signal()=TALKIN
    },
    TALKIN{
        override fun signal()=WATTING
    };
    abstractfunsignal():anonys
}
fun main(args:Array<String>){
    println(rainbow.values().joinToString())
    println(corour.GREEN)
    println(corour.valueOf("BLUE").name)
    println(corour.valueOf("BLUE").ordinal)// 序号
    println(corour.values().joinToString{it.name+":"+it.rgb})
    println(corour.valueOf("BLUE").rgb)
println(anonys.values().joinToString{it.name})
}
```

运行结果如下：

```
赤，橙，黄，绿，青，蓝，紫
GREEN
BLUE
2
RED:16711680，GREEN:65280，BLUE:255
255
WATTING，TALKIN
```

就像在 Java 中一样，Kotlin 中的枚举类可以通过合成方法列出定义了的枚举常量，而且通过其名字来获取枚举常量。

```
EmumClass.valueof(value:String):EmumClass
EmumClass.value():Array<EmumClass>
```

如果指定的名称和类定义的任何枚举常量不匹配，则 valueof()方法会抛出异常。在 corour 类中没有“no”枚举常量，这时候输出的结果就是：

```
println(corour.valueOf("no").rgb)
// error:抛出 IllegalArgumentException 异常:Exceptioninthread"main"
java.lang.IllegalArgumentException:Noenumconstantjqiang.hight.corour
.no
```

也可以使用 enumValues<T>()和 enumValuesof<T>()函数以泛型方式访问类的常量。代码如下：

```
println(enumValues<rainbow>().joinToString{it.name})
    println(enumValueOf<corour>("RED").rgb)
```

每一个枚举常量都具有在枚举类声明中获取其名称和位置的属性。代码如下：

```
val name:String
val ordinal:Int
```

5.3.7　对象表达式和对象声明

有时候在创建子类时只需要进行小小的改动，这样没有太大的问题，在 Java 中使用匿名类来处理这种情况，但是在 Kotlin 中可以使用对象表达式或对象声明来处理这种发生细微变化的类。

1．对象表达式

如果超类型有一个构造函数，则必须传递适当的参数给它。多个超类型可以

由跟在冒号后面的以逗号分隔的列表指定。

【例 5.15】创建一个类 A，然后在 main 函数中创建一个与类 A 相关的对象表达式。

```
package jqiang.hight
open class A(x:Int){
    publicopen valy:Int=x
}
fun main(args:Array<String>){
    var ab:A=object:A(10){
        overridevaly=15
    }
}
```

任何时候，如果只需要“一个对象”而已，并不需要特殊的超类型，那么可以简单地写成这样：

```
val ab=object{
    val x:Int=10
    val y:Int=15
}
println(ab.x+ab.y)
```

运行结果如下：

```
25
```

总结：匿名对象可以用作只在本地和私有作用域中声明的类型。如果将匿名对象作为公有函数的返回类型或者公有属性的类型，那么该函数或属性的实际类型就是匿名对象声明的超类型，如果没有声明任何超类型，那么就是 Any，将无法访问在匿名对象中添加的成员。

2．对象声明

单例是一种非常有用的开发模式，在 Kotlin 中使用单例模式来声明一个对象会变得更加容易。

声明对象和声明变量一样，对象声明并不是一个表达式，所以不能使用赋值语句进行赋值操作。

声明对象一般是在 object 后面加上对象名称。

【例 5.16】创建一个 Preson 类和对象 Playroll，使用单例模式添加成员属性和方法。

```
package jqiang.hight
    class Preson()
    object Payroll{
        val all=arrayListOf<Preson>()
        fun show(){
            for(preson in all){
                println(preson.toString())
            }
        }
    }
    fun main(args:Array<String>){
        Payroll.all.add(Preson())
        Payroll.show()
    }
```

总结：声明对象时要使用 object 关键字；对象声明不是一个表达式，所以不能使用赋值语句直接赋值给对象，但是可以嵌套到其他对象声明或者非内部类中。

对象表达式和对象声明之间的区别：对象表达式是立即执行的；对象声明在第一次访问时被懒惰地初始化，在第二次访问时速度和表达式相似。当对应的类被加载（解析）时，伴随着对象被初始化，与 Java 静态初始化器的语义相匹配。

3．伴生对象

Kotlin 中的类不能有静态成员，Java 中的 static 关键字不是 Kotlin 编程语言的一部分，Kotlin 可以通过包级函数和对象声明来实现 static 功能。在大多情况下，一般推荐使用顶层函数，但是顶层函数不能访问类中的 private 成员

这时候我们编写一个函数，这个函数不需要通过类实例进行调用，但是需要访问类的内部，则可以把这个函数作为内部对象声明成员。

【例 5.17】创建一个最简单的对象 Resource，然后分析它的执行过程。

```
package jqiang.hight
object Resource{
    val name="Kotlin"
    fun show(){
        println("Hello")
    }
}
```

这样我们就能得到一个单例类。反编译后生成的代码如下：

```
public final classe Resource{
{
```

```
    public static final Resource INSTANCE;;
    privatec static final String name=="Kotlin";;
    static{
        newResource();
    }
    private Resource(){
        INSTANCE=this;
    }
    public static final StringgetName(){
        return name;
    }
    public static final show(){
        System.out.println("Hello")
    }
}
```

通过这个简单的对象我们不难发现，常量 name 和 show()方法虽然不是包级的成员，但是通过反编译之后变成静态成员。利用这个特性，Kotlin 允许在类中使用 companion object 创建伴生对象，用伴生对象的成员来代替静态成员。

【例 5.18】创建一个 ObjectCom 类，在类中创建一个伴生对象 Factory，在伴生对象中只有一个 create()方法。

```
package jqiang.hight
class ObjectCom{
    companion object Factory{
        fun create():ObjectCom=ObjectCom()
    }
}
```

可以使用简单的类名作为限定符来调用伴生对象的成员。代码如下：

```
fun main(args:Array<String>){
    val objectCom=ObjectCom.create()
}
```

在使用 companion 关键字时，伴生对象 Factory 是可以省略不写的。代码如下：

```
class ObjectCom{
    companion object{
        funcreate():ObjectCom=ObjectCom()
    }
}
```

伴生对象的成员看起来像其他编程语言中的静态成员，在运行时它们仍然是

对象的实例成员，因此伴生对象也是可以实现接口的。实现代码如下：

```
interface Factory<T>{
    funcreate():T
}
class MyClass{
    companion object:Factory<MyClass>{
        override fun create():MyClass=MyClass()
    }
}
```

5.3.8　密封类

当值只是限定类型组中的一种时，可使用密封类表示受限的类层次。从某种意义上说，密封类是枚举类的扩展类型：枚举类型值的限定集合，但枚举类型只存在一个实例，而密封类的子类可以包含状态的多个实例。

声明密封类时，需要在类名前面加 sealed 修饰符。密封类可以有子类，但所有子类都必须在密封类对应的文件中声明。在 Kotlin 1.1 版本中，数据类可继承其他类，包括密封类，继承密封类的子类可以存放在任何地方。密封类的主要好处就是可以在 when 表达式中使用。如果能够覆盖所有条件，则不用在条件语句中添加 else 代码块。

声明密封类的一般形式如下：

```
sealed class 类名
```

【例 5.19】创建一个密封类 Expr 和两个继承自密封类的数据类 Const 和 Sum，并且创建一个对象继承自 Expr 类。

```
sealed class Expr
data class Const(valnumber:Double):Expr()
data class Sum(vale1:Expr, vale2:Expr):Expr()
object NotANumber:Expr()
```

密封类的好处在于使用 when 表达式，如果能覆盖所有的情况，则无须添加 else 条件语句。使用方式如下：

```
fun eval(expr:Expr):Double=when(expr){
    isConst->expr.number
    isSum->eval(expr.e1)+eval(expr.e2)
    NotANumber->Double.NaN
}
```

5.3.9 抽象类

抽象类往往用来表征对问题领域进行分析、设计得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。比如，在一个图形编辑软件的分析、设计过程中，会发现问题领域存在着圆、三角形这样一些具体的概念，它们是不同的，但是它们又都属于形状这个概念，形状概念在问题领域并不是直接存在的，它就是一个抽象概念。而正是因为抽象概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能够实例化的。

在编程语言中，通常使用 abstract 修饰的类就是抽象类。在 C++中，含有纯虚函数的类称为抽象类，它不能生成对象；在 Java 中，含有抽象方法的类称为抽象类，它同样不能生成对象。

抽象类是不完整的，它只能用作基类。在面向对象方法中，抽象类主要用来进行类型隐藏和充当全局变量的角色。

在面向对象编程中，所有的对象都是通过类来描绘的；但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，那么这样的类就是抽象类。

在 Kotlin 中，抽象类可以不用实现，也不需要使用 open 关键字对该类进行标注。

抽象类的一般格式如下：

```
abstract class 类名{}
```

【例 5.20】使用一个抽象成员覆盖一个非抽象的开放成员。

```
open classBase{
    open funf(){}
}
abstract class Derived:Base(){
    override abstract fun f()
}
```

抽象类和普通类的区别如下：

- 抽象方法必须为 public 或 protected。如果为 private，则不能被子类继承，子类无法实现该抽象方法。不写修饰符，则默认为 public。
- 抽象类不能用于创建对象。
- 如果一个子类继承自抽象类，则子类必须实现父类的抽象方法；如果子类没有实现父类的抽象方法，就必须将子类也定义为一个抽象类。如果抽象

类中含有抽象属性，那么在实现子类时必须将抽象属性初始化，除非子类也是一个抽象类。

5.3.10　接口的使用

Kotlin 接口（Interface）是一些方法特征的集合，这些方法特征来自于具体方法。接口只有方法的特征，而没有方法的实现，因此这些方法在不同的地方被实现时，可以具有完全不同的行为。在 Kotlin 中，接口还可以定义 public 变量。

接口把方法的特征和方法的实现分割开。这种分割体现为接口常常代表一个角色（role），它包装与该角色相关的操作和属性，而实现这个接口的类便扮演这个角色的演员（类）。一个角色（接口）可以由不同的演员（类）来演，而不同的演员（类）之间除扮演一个共同的角色（接口）之外，并不要求有任何其他的共同之处。

继承特性简化了对象、类的创建，提高了代码的可复用性。Kotlin 的接口和 Java 8 的接口有点类似，既包含了抽象方法的声明，又包含了实现。与抽象类不同的是，接口无法保存状态，它可以有属性，但必须声明为抽象的或提供访问器实现。

使用 interface 关键字定义接口，一般格式如下：

```
interface 接口名{}
```

子类或对象可以通过继承的方法来实现接口，如果要实现多个接口，每个接口之间用逗号隔开，而且在接口中没有完成的方法要在子类中完成。

在编程过程中有很多地方会用到接口，比如封装一个上传类，这时候我们要根据上传到不同的位置来封装不同的类。这样做很麻烦，所以做成接口的形式，一个接口可以实现很多方法，从接口中调用不同的方法和实现不同的功能。比如数据库连接，mysql 类具备增、删、查、改功能，mssql 类也同样具备这些功能，这时我们就可以使用一个接口，使 mysql 和 mssql 类继承该接口。

接口的特点如下：

（1）Kotlin 接口中的成员变量默认都是 public、open 类型的（都可省略），必须被显式初始化，即接口中的成员变量为常量

（2）Kotlin 接口中的方法默认都是 public、open 类型的（都可省略），没有方法体，不能被实例化。

（3）在 Kotlin 接口中只能包含 public、open 类型的成员变量和成员方法。

（4）接口中没有构造方法，不能被实例化。

（5）一个接口不能实现（Implement）另一个接口，但是可以继承多个其他的接口。

（6）Kotlin 接口必须通过类来实现它的抽象方法。

（7）当类实现了某个 Kotlin 接口时，它必须实现接口中的所有抽象方法；否则，这个类必须声明为抽象类。

（8）不允许创建接口的实例（实例化），但允许定义接口类型的引用变量，该引用变量引用实现了这个接口的类的实例。

（9）一个类只能继承一个直接父类，但可以实现多个接口，间接实现了多继承。

【例 5.21】以数据库为例，数据库的一般流程是：连接数据库服务器，选择数据库，执行数据库语句，关闭结果集。

```
   package jqiang.entrust
   interface database{
      public var host:String
      var port:Int
      var database:String
      var user:String
      var password:String
      funconnect()
      funoperate()
      funclose()
      funover(){
         println("成功关闭数据库")
      }
   }
   class mysql(override var host:String="127.0.0.1", override var
port:Int=3306, override var database:String="demo", override var
user:String="sa", override var password:String="123"):database{

      override fun connect(){
         println("开始连接数据库...")
      }
      override fun operate(){
         println("操作${database}数据库...")
      }
      override fun close(){
         println("正在关闭${database}数据库连接...")
      }
```

```
    }
    fun main(args:Array<String>){
        var mysql=mysql()println("您好，您的数据库地址是${mysql.host}:
${mysql.port}，您的数据库是${mysql.database}，用户名：${mysql.user}，密码：
${mysql.password}")
        println(mysql.connect())
        println(mysql.operate())
        println(mysql.close())
        println(mysql.over())
    }
```

总结：在接口中定义的属性且无赋值语句，属于抽象属性；在接口中定义的方法且无方法体，属于抽象方法。抽象属性和抽象方法一定要在子类在重写；在接口中定义的已赋值的属性或拥有方法体的方法，属于已实现的属性或已实现的方法。

如果一个子类继承了两个接口，那么就需要使用 super<T>.属性或方法来继承其父类的属性和方法。例如：

```
    interface A{
        funfoo(){print("A")}// 已实现
        funbar()// 未实现，没有方法体，是抽象的
    }

    interface B{
        funfoo(){print("B")}// 已实现
        funbar(){print("bar")}// 已实现
    }

    classC:A{
        override fun bar(){print("bar")}// 重写
    }
    classD:A，B{
        override fun foo(){
            super<A>.foo()
        super<B>.foo()
        }
        override fun bar(){
            super<B>.bar()
        }
    }
```

在上面代码中，接口 A 和 B 都定义了方法 foo()和 bar()，两者都实现了 foo()，

B 实现了 bar()。因为 C 是一个实现了 A 的具体类，所以必须要重写 bar()并实现这个抽象方法。

然而，如果从 A 和 B 派生 D，则需要实现多个接口继承的所有方法，并指明 D 应该如何实现它们。这个规则既适用于继承单个实现（bar()）的方法，也适用于继承多个实现（foo()）的方法。

5.3.11 泛型

泛型，即“参数化类型”。一提到参数，就会想到定义方法时有形参，然后调用此方法时传递实参。那么怎么理解参数化类型呢？顾名思义，参数化类型就是将原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

与 Java 中的泛型一样，Kotlin 中的泛型的本质就是参数化类型，也就是所操作的数据类型被指定为一个参数。这种类型参数可以使用在类、接口和方法的创建中，分别称为泛型类、泛型接口和泛型方法。

创建泛型的一般形式如下：

```
classBox<T>(t:T){var value=t}
```

实例化泛型的形式如下：

```
val box:Box<Int>=Box<Int>(1)
val box=Box(1)
```

在 Java 中泛型是最棘手的是通配符类型，在 Kotlin 中对应有两种情况：位置差异和类型投影

1．声明位置差异

假设有一个泛型接口 Source<T>，该接口中不存在任何以 T 作为参数的方法，只是方法返回 T 类型值。那么在 Source<Object>类型的变量中存储 Source<String>实例的引用是极为安全的，对象没有被使用，它的方法也是可以被调用的。但是在 Java 中无法实现这一点，它会禁止这种操作。可以注解 Source 的类型参数 T，以确保它仅从成员返回（生成）Source<T>。为此，Kotlin 提供了 out 修饰符：

```
abstract class Source<out T>{
    abstractfunnextT():T
}
fun demo(strs:Source<String>){
```

```
    val objects:Source<Any>=strs
}
```

out 修饰符称为型变注解，并且由于它是在类型参数声明处提供的，所以这里就被称之为声明处型变。这与 Java 的使用处型变相反，其类型用途通配符使得类型协变。

```
abstract class Comparable<inT>{
    abstract fun compareTo(other:T):Int
}
fund emo(x:Comparable<Number>){
    x.compareTo(1.0)// 1.0 拥有类型 Double，它是 Number 的子类型
// 因此，我们可以将 x 赋值给类型为 Comparable<Double>的变量
    val y:Comparable<Double>=x//OK!
}
```

2．类型投影

将类型参数 T 声明为 out 非常方便，并且能避免使用处子类型化的麻烦。但是有些类实际上不能限制为只返回 T。比如 Array 类：

```
class Array<T>(valsize:Int){
    fun get(index:Int):T{///*……*/}
    fun set(index:Int, value:T){///*……*/}
}
```

该类在 T 上既不能是协变的，也不能是逆变的，这导致不灵活。考虑下述函数：

```
fun copy(from:Array<Any>, to:Array<Any>){
    assert(from.size==to.size)
    for(iinfrom.indices)
    to[i]=from[i]
}
```

这个函数用于将一个数组复制到另一个数组。我们尝试在实践中应用它：

```
val ints:Array<Int>=arrayOf(1, 2, 3)
val any=Array<Any>(3)
copy(ints, any)// 错误：期望(Array<Any>, Array<Any>)
```

如果使用 copy()函数将传入 from 的参数为 Int 类型的数组复制到一个超类型的数组中，将会抛出 ClassCastException 异常。现在，我们唯一要确保的是 copy()函数不会做出对不符合类型的数据进行复制的事情，阻止它写到 from，可以这样做：

```
fun copy(from:Array<outAny>, to:Array<Any>){//…}
```

这里发生的事情称为类型投影。from 是一个受限制的（投影的）数组，我们只可以调用返回类型为类型参数 T 的方法，这意味着只能调用 get()。这就是使用处型变的用法，并且对应于 Java 的 Array<?extendsObject>。

也可以使用 in 投影一种类型：

```
fun fill(dest:Array<inString>, value:String){//…}
```

Array<inString>对应于 Java 的 Array<?superString>，也就是说，可以传递一个 CharSequence 或 Object 数组给 fill()函数。

3．星投影

有时候我们对类型的参数一无所知，但是又想安全地使用它们，这时所使用的方法是定义泛类型的参数投影，被称为星投影。

Kotlin 为此提供了所谓的星投影语法：

对于 Foo<out T>，其中 T 是一个具有上界 TUpper 的协变类型参数，Foo<*>等价于 Foo<out TUpper>。这意味着当 T 未知时，可以安全地从 Foo<*>读取 TUpper 的值。

对于 Foo<in T>，其中 T 是一个逆变类型参数，Foo<*>等价于 Foo<in-Nothing>。这意味着当 T 未知时，没有任何方法可以安全地写入。

对于 Foo<T>，其中 T 是一个具有上界 TUpper 的不型变类型参数，Foo<*>在读取值时等价于 Foo<outTUpper>，而在写值时等价于 Foo<inNothing>。

如果泛型类型具有多个类型参数，那么每个类型参数都可以单独投影。例如，如果类型被声明为 interface Function<inT, outU>，则可以考虑进行如下星投影：

- Function<*，String>，表示 Function<inNothing，String>；
- Function<Int，*>，表示 Function<Int，outAny?>；
- Function<*，*>，表示 Function<inNothing，outAny?>。

注意：星投影非常像 Java 的原始类型，但是它安全、实用。

5.4 委托和委托属性

对于一些常见的属性类型，虽然在每次需要的时候可以手动实现它们，但是

如果能只实现一次并放入一个库中则会更好。例如：

（1）延迟属性（Lazy Property），其值只在首次访问时计算。

（2）可观察属性（Observable Property），监听器会收到有关此属性变更的通知。

（3）把多个属性存储在一个映射（Map）中，而不是存储在单独的字段中。

为了涵盖这些（以及其他）情况，Kotlin 支持委托属性：

```
class Example{var p:StringbyDelegate()}
```

创建委托属性的一般形式如下：

```
val/var <属性名>:<类型>by<表达式>
```

by 后面的表达式是委托，因为属性对应的 get()（和 set()）会被委托给它的 getValue()和 setValue()方法。委托属性不必实现任何接口，但是需要提供一个 getValue()函数（和 setValue()——对于 var 属性）。

```
class Delegate{
    operator fun getValue(this Ref:Any?, propert y:KProperty<*>):
String{
        return"$thisRef, thankyoufordelegating'${property.name}'tome!"
    }
    operatorfunsetValue(thisRef:Any? , property:KProperty<*> ,
value:String){
        println("$valuehasbeenassignedto'${property.name}
in$thisRef.'")
    }
}
```

当我们从委托到一个 Delegate 实例的 p 读取时，将调用 Delegate 中的 getValue()函数，所以它的第一个参数用于读出 p 的对象，第二个参数保存了对 p 自身的描述（可以取它的名字)。例如：

```
vale=Example()
println(e.p)
```

输出结果如下：

```
Example@33a17727, thankyoufordelegating'p'tome!
```

类似的，当给 p 赋值时，将调用 setValue()函数。它的前两个参数与 getValue()相同，第三个参数保存了将要被赋予的值。

```
e.p="NEW"
```

输出结果如下：

```
NEWhasbeenassignedto 'p'inExample@33a17727.
```

1．标准委托

在 Kotlin 的标准库中，为有用的委托提供了很好的工厂方法。

2．延迟属性

lazy()是一个接受 Lambda 表达式并返回 Lazy<T>实例的函数，所返回的实例可以作为实现延迟属性的委托：第一次调用 get()会执行已传递给 lazy()的 Lambda 表达式并记录结果，后续调用 get()只是返回记录的结果。

```
package jqiang.entrust
val lazyValue:Stringbylazy{
    println("computed!")
    "Hello"
}
fun main(args:Array<String>){
    println(lazyValue)
    println(lazyValue)
}
```

运行结果如下：

```
computed!
Hello
Hello
```

在默认情况下，对于 lazy 属性的求值是同步锁的（Synchronized）——该值只在一个线程中计算，并且所有线程会看到相同的值。如果初始化委托的同步锁不是必需的，这样多个线程可以同时执行，那么将 LazyThreadSafetyMode.PUBLICATION 作为参数传递给 lazy()函数。如果确定初始化总是发生在单个线程中，那么可以使用 LazyThreadSafetyMode.NONE 模式，它不会保证任何线程的安全，也没有相关的开销。

3．可观察属性

Delegates.observable()接受两个参数：初始值和修改时处理程序（Handler）。每当给属性赋值时就会调用该处理程序（在赋值后执行）。它有三个参数，分别是被赋值的属性、旧值和新值。

```
package jqiang.entrust
import kotlin.properties.Delegates
class User{
    var name:StringbyDelegates.observable("<noname>"){
        prop, old, new->
        println("$old->$new")
    }
}
fun main(args:Array<String>){
    val user=User()
    user.name="first"
    user.name="second"
}
```

运行结果如下：

```
<noname>->first
first->second
```

如果想截获一个赋值并“否决”它，就使用 vetoable()取代 observable()，在属性被赋新值生效之前会调用传递给 vetoable()的处理程序。

4．把属性存储在映射中

把不固定的值存放在 Map 集合中，可以随时进行调取，通过 map()中的 key 值获取对应的 value 值。在这种情况下，可以使用映射实例自身作为委托来实现委托属性。

```
class User(valmap:Map<String, Any?>){
    val name:Stringbymap
    val age:Intbymap
}
```

在这个例子中，构造函数接受一个映射参数：

```
val user=User(mapOf("name"to"jqiang", "age"to25))
```

委托属性会从这个映射中取值（通过字符串键——属性的名称）：

```
println(user.name)
println(user.age)
```

如果把只读的 Map 换成 MutableMap，那么也适用于 var 属性。

```
class MutableUser(val map:MutableMap<String, Any?>){
    var name:Stringbymap
```

```
    var age:Intbymap
}
```

5．局部委托属性

可以将局部变量声明为委托属性。例如，可以将一个局部变量惰性初始化：

```
fun example(computeFoo:()->Foo){
    val memoizedFoobylazy(computeFoo)
    if(someCondition&&memoizedFoo.isValid()){
        memoizedFoo.doSomething()
    }
}
```

memoizedFoo 变量只会在第一次访问时计算。如果 someCondition 失败，那么该变量根本不会计算。

6．属性委托要求

对于一个只读属性（即用 val 声明的），委托必须提供一个名为 getValue 的函数，这个函数必须返回与属性相同的类型（或其子类型）。该函数接受以下参数：

- thisRef——必须与属性的所有者类型（对于扩展属性，指被扩展的类型）相同或者是它的超类型。
- property——必须是类型 KProperty<*>或者其超类型。

对于一个可变属性（即用 var 声明的），委托必须额外提供一个名为 setValue 的函数，getValue()或/和 setValue()函数可以通过委托类的成员函数提供或者由扩展函数提供。当需要委托属性到原本未提供的这些函数的对象时，后者会更便利。这两个函数都需要使用 operator 关键字来标识。

该函数接受以下参数：

- thisRef——同 getValue()。
- property——同 getValue()。
- newvalue——必须和属性的类型相同或者是它的超类型。

委托类可以实现包含所需 operator 方法的 ReadOnlyProperty 或 ReadWriteProperty 接口之一。这两个接口是在 Kotlin 的标准库中声明的。

```
interface ReadOnlyProperty<inR, outT>{
    operatorfungetValue(thisRef:R, property:KProperty<*>):T
}
interface ReadWriteProperty<inR, T>{
```

```
    operatorfungetValue(thisRef:R, property:KProperty<*>):T
    operatorfunsetValue(thisRef:R, property:KProperty<*>, value:T)
}
```

7．翻译规则

在每个委托属性的实现背后，Kotlin 编译器都会生成辅助属性并委托给它。例如，对于属性 prop，生成隐藏属性 prop $delegate，而编译器中的代码只是简单地委托给这个辅助属性。

```
class C{
    var prop:TypebyMyDelegate()
}
// 这段是由编译器生成的相应代码
class C{
    private val prop$delegate=MyDelegate()
    var prop:Type
    get()=prop$delegate.getValue(this, this::prop)
    set(value:Type)=prop$delegate.setValue(this, this::prop, value)
}
```

Kotlin 编译器在参数中提供了关于 prop 的所有必要信息，其中第一个参数 this 引用到外部类 C 的实例；第二个参数 this::prop 是 KProperty 类型的反射对象，该对象描述 prop 自身。

8．提供委托

通过定义 provideDelegate 操作符，可以扩展创建属性实现所委托对象的逻辑。如果在 by 右边所使用的对象将 provideDelegate 定义为成员或扩展函数，那么会调用该函数来创建属性委托实例。

provideDelegate 的一个可能的使用场景是在创建属性时（而不仅仅在其 getter 或 setter 中）检查属性的一致性。

例如，如果要在绑定之前检查属性名称，则可以这样写：

```
class ResourceLoader<T>(id:ResourceID<T>){
    operator fun provideDelegate(
            thisRef:MyUI,
            prop:KProperty<*>
    ):ReadOnlyProperty<MyUI, T>{
        checkProperty(thisRef, prop.name)
    }
    private fun checkProperty(thisRef:MyUI, name:String){…}
```

```
    }
    fun<T>bindResource(id:ResourceID<T>):ResourceLoader<T>{…}
    classMyUI{
        val image by bind Resource(ResourceID.image_id)
        val text by bind Resource(ResourceID.text_id)
    }
```

provideDelegate 的参数含义与 getValue 相同，这里不再赘述。

在创建 MyUI 实例期间，为每个属性调用 provideDelegate 函数，并立即执行必要的验证。

为了实现相同的功能，必须显式传递属性名。这样做不是很方便。

```
    // 检查属性名称，不使用 provideDelegate 的功能
    class MyUI{
        val image by bind Resource(ResourceID.image_id, "image")
        val text by bind Resource(ResourceID.text_id, "text")
    }
    fun<T>MyUI.bindResource(
            id:ResourceID<T>,
            propertyName:String
    ):ReadOnlyProperty<MyUI, T>{
        checkProperty(this, propertyName)
    }
```

在所生成的代码中，会调用 provideDelegate 函数来初始化辅助属性 prop$delegate。比较属性声明 valprop:TypebyMyDelegate()生成的代码与上面（当 provideDelegate 函数不存在时）所生成的代码：

```
    class C{
        var prop:TypebyMyDelegate()
    }
    // 这段代码是当 provideDelegate 的功能可用时
    // 由编译器生成的
    class C{
        // 调用 provideDelegate 来创建额外的委托属性
        private val prop$delegate=MyDelegate().provideDelegate(this,
this::prop)
        val prop:Type
            get()=prop$delegate.getValue(this, this::prop)
    }
```

provideDelegate 函数只影响辅助属性的创建，并不会影响为 getter 或 setter 生成的代码。

5.5　错误与异常

在程序开发过程中，涉及程序的错误包括两种：编译时错误和运行时错误。首先应对源程序进行编译，在编译过程中会发现程序中存在的语法错误，编译程序能够检查出这些错误，这是正常的，不属于异常；而通过编译之后，在 JVM 中运行程序时可能因为某种原因以及某种事件的出现，导致程序产生错误无法运行，这就属于异常（Exception）。比如，对两个数进行除法运算时，如果除数为 0，那么程序将会被中止，这种现象在编译时不会出现任何错误，只有当程序执行到含有除法运算的语句时才会发生错误。为了使程序更健壮，要对异常进行处理（称之为异常处理）。

异常是导致程序中断运行的一种指令流，如果不对异常进行正确的处理，则可能会导致程序中断运行，造成不必要的损失。所以，在程序设计中要考虑可能发生的各种异常，并做出正确的处理，这样才能保证程序正常运行。在 Kotlin 中，一切异常都秉承面向对象的设计思想，所有的异常都是以类和对象的形式出现的，除 Kotlin 类库中已经提供的各种异常类外，用户可以根据自己的需求自定义异常类。

在 Kotlin 中所有的异常类都是 Throwable 类的子孙类，每种异常都有消息、堆栈回溯信息和可能的原因。

使用 throw 表达式来抛出异常：

```
throw MyException("This is an exception!")
```

使用 try 表达式来捕获异常：

```
try { // 一些代码 }
catch (e: SomeException) { // 处理程序 }
finally { // 可选的 finally 语句块 }
```

（1）try 语句块：表示要尝试运行的代码，try 语句块中的代码受异常监控，当该代码发生异常时，会抛出异常对象。

（2）catch 语句块：会捕获 try 代码块中发生的异常，并进行异常处理。catch 语句带一个 Throwable 类型的参数，表示可捕获的异常类型。当 try 语句块中出现异常时，catch 会捕获到所发生的异常，并和自己的异常类型进行匹配，若匹配，则执行 catch 语句块中的代码，并将 catch 参数指向所抛出的异常对象。可以有多个 catch 语句块，用来匹配其中的一种异常，一旦匹配，就不再尝试匹配其他的

catch 语句块。通过异常对象可以获取异常发生时完整的 JVM 堆栈信息，以及异常信息和异常发生的原因等。

（3）finally 语句块：紧跟在 catch 语句块的后面，这个语句块总是在方法返回前执行，而不管 try 语句块是否发生异常。

注意：

- try、catch 和 finally 这三个语句块均不能单独使用，三者可以组成 try...catch...finally、try...catch、try...finally 三种结构。catch 语句块可以有一个或多个，finally 语句块最多只能有一个。
- try、catch 和 finally 这三个语句块中的变量的作用域为对应的语句块内部，分别独立而不能相互访问。如果要在这三个语句块中都可以访问，则需要将变量定义在这些块的外面。
- 当有多个 catch 语句块时，只会匹配其中的一种异常类型并执行这个 catch 语句块，而不会再执行其他的 catch 语句块，并且匹配 catch 语句块的顺序是由上到下的。
- 在 try 表达式中可以有 0 个或多个 catch 语句块，finally 语句块可以省略，但是 catch 或 finally 至少要有一个与 try 配对出现。

5.5.1 自定义异常类

在 Kotlin 的标准库中封装的异常类型，不可能会预见所有的异常情况，此时可以自定义异常类来表示程序中可能出现的特定问题。

如果要自定义异常，就必须继承现有的异常类，一般都继承与其异常情况相似的类。自定义异常类最简单的方法就是使用编辑器产生默认的构造方法，这样做简单而有效。

【例 5.22】自定义空值异常类。

```
package jqiang.Throwable
fun main(args: Array<String>) {
    var b:String?=null
    try {
        b?.length?:throw MyExcepttion("空值")
    }catch (e:MyExcepttion){
        println("e:${e.message}")
    }catch (N:NullPointerException){
        println("N:${N.message}")
```

```
        }finally {
            println("finally")
        }
    }
    class MyExcepttion(override val message: String) : Throwable() {}
```

运行结果如下：

```
e:空值
finally
```

5.5.2　try 表达式

try 是一个表达式，即它可以有一个返回值。

```
    val a: Int? = try { parseInt(input) } catch (e: NumberFormatException)
{ null }
```

try 表达式的返回值，要么是 try 语句块内最后一个表达式的值，要么是 catch 语句块内最后一个表达式的值。finally 语句块的内容不会影响 try 表达式的结果值。

5.6　包

为了更好地组织类，Kotlin 提供了包（Package）机制，包是类的容器，用于分隔类名空间。如果没有指定包名，那么所有的示例都属于一个默认的无名包。

包的作用如下：

（1）把功能相似或相关的类或者接口组织在同一个包中，以方便查找和使用类。

（2）如同文件夹一样，包也采用树形的存储方式。在同一个包中类名不可以相同，在不同的包中类名可以相同，当同时调用不同包中的具有相同类名的类时，应该加上包名加以区别。因此，使用包可以避免命名冲突。

（3）包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类。

包语句的语法格式如下：

```
    package jqiang.bar
```

它的路径应该是 jqiang/bar/目录。包的作用是把不同的 Kotlin 程序分类保存，以方便被其他 Kotlin 程序调用。

在 Kotlin 中默认导入的包有：Kotlin.*、Kotlin.annotation.*、Kotlin.collections.*、Kotlin.comparisons.* (since 1.1)、Kotlin.io.*、Kotlin.ranges.*、Kotlin.sequences.*、Kotlin.text.*。

使用 Kotlin 创建不同的开发平台需要自动导入不同的包，比如：

JVM——java.lang.*和 Kotlin.jvm.*

JavaScript——Kotlin.js.*

为了能够使用某一个包的成员，需要在 Kotlin 程序中明确导入该包。使用 import 语句可完成此功能。

可以直接导入单个包，例如：

```
import jqiang.Bar
```

也可以导入一个作用域下的所有内容（包、类、对象等）：

```
import jqiang.*
```

如果名字冲突，则可以使用 as 关键字在本地重命名冲突项来消除歧义：

```
import jqiang.Bar
import jqiang.Bar as bBar
```

5.6 本章小结

本章主要介绍了面向对象的基本概念和 Kotlin 的特性，如数据类、抽象类、接口等。虽然本章对面向对象介绍得很全面、详细，但是想要真正明白面向对象的思想，就必须多动手实践，多动脑思考，注意平时要多积累等，相信读者通过自己的努力，一定会有所突破。

第 6 章

Kotlin 互操作

互操作就是在 Kotlin 中可以调用其他编程语言的接口，只要它们开放了接口，Kotlin 就可以调用其成员属性和成员方法，这是其他编程语言所无法比拟的。同时，在进行 Java 编程时也可以调用 Kotlin 中的 API 接口。

本章主要内容：

- Kotlin 与 Java 互操作
- Kotlin 与 JavaScript 互操作

6.1 Kotlin 与 Java 互操作

6.1.1 Kotlin 调用 Java

Kotlin 在设计时就考虑了与 Java 的互操作性。可以从 Kotlin 中自然地调用现有的 Java 代码，在 Java 代码中也可以很顺利地调用 Kotlin 代码。

【例 6.1】在 Kotlin 中调用 Java 的 Util 的 list 库。

```
packagejqiang.Mutual.Kotlin
importjava.util.*
fundemo(source:List<Int>){
    vallist=ArrayList<Int>()
    for(iteminlist){
        list.add(item)
    }
    for(iin0..source.size-1){
        list[i]=source[i]
    }
}
```

基本的互操作行为如下：

1．属性读写

Kotlin 可以自动识别 Java 中的 getter/setter；在 Java 中可以过 getter/setter 操作 Kotlin 属性。

【例 6.2】自动识别 Java 中的 getter/setter。

```
packagejqiang.Mutual.Kotlin
importjava.util.*
funmain(args:Array<String>){
    valcalendar=Calendar.getInstance()
    println(calendar.firstDayOfWeek)
    if(calendar.firstDayOfWeek==1){// 调用 getFirstDayOfWeek()方法
        calendar.firstDayOfWeek=2// 调用 setFirstDayOfWeek()方法
    }
    println(calendar.firstDayOfWeek)
}
```

遵循 Java 约定的 getter 和 setter 方法（名称以 get 开头的无参数方法和以 set 开头的单参数方法）在 Kotlin 中表示为属性。如果 Java 类只有一个 setter，那么它在 Kotlin 中不会作为属性可见，因为 Kotlin 目前不支持只写（set-only）属性。

2．空安全类型

Kotlin 的空安全类型的原理是，Kotlin 在编译过程中会增加一个函数调用，对参数类型或者返回类型进行控制，开发者可以在开发时通过注解@Nullable 和@NotNull 方式来弥补 Java 中空值异常。

Java 中的任何引用都可能是 null，这使得 Kotlin 对来自 Java 的对象进行严格的空安全检查是不现实的。Java 声明的类型在 Kotlin 中称为平台类型，并会被特别对待。对这种类型的空检查要求会放宽，因此对它们的安全保证与在 Java 中相同。

【例 6.3】空值实例。

```
vallist=ArrayList<String>()//非空（构造函数结果）
list.add("Item")
val size=list.size()//非空（原生 Int）
Val item=list[0]//推断为平台类型（普通 Java 对象）
```

当调用平台类型变量的方法时，Kotlin 不会在编译时报告可空性错误，但是在运行时调用可能会失败，因为空指针异常。

```
item.substring(1)//允许，如果 item==null 可能会抛出异常
```

平台类型是不可标识的，这意味着不能在代码中明确地写下它们。当把一个平台值赋给一个 Kotlin 变量时，可以依赖类型推断（该变量会具有所推断出的平台类型，如上例中 item 所具有的类型），或者选择我们所期望的类型（可空的或非空类型均可）。

```
val nullable:String?=item//允许，没有问题
Val notNull:String=item//允许，运行时可能失败
```

如果选择非空类型，编译器会在赋值时触发一个断言，这样可以防止 Kotlin 的非空变量保存空值。当把平台值传递给期待非空值等的 Kotlin 函数时，也会触发一个断言。总的来说，编译器尽力阻止空值通过程序向远传播（由于泛型的原因，有时这不可能完全消除）。

3．返回 void 的方法

如果在 Java 中返回 void，那么 Kotlin 返回的就是 Unit。如果在调用时返回 void，那么 Kotlin 会事先识别该返回值为 void。

4．注解的使用

@JvmField 是 Kotlin 和 Java 互相操作属性经常遇到的注解；@JvmStatic 是将对象方法编译成 Java 静态方法；@JvmOverloads 主要是 Kotlin 定义默认参数生成重载方法；@file:JvmName 指定 Kotlin 文件编译之后生成的类名。

5．NoArg 和 AllOpen

数据类本身属性没有默认的无参数的构造方法，因此 Kotlin 提供一个 NoArg 插件，支持 JPA 注解，如@Entity。AllOpen 是为所标注的类去掉 final，目的是为了使该类允许被继承，且支持 Spring 注解，如@Componet；支持自定义注解类型，如@Poko。

6．泛型

Kotlin 中的通配符“*”代替 Java 中的“？”；协变和逆变由 Java 中的 extends 和 super 变成了 out 和 in，如 ArrayList<outString>；在 Kotlin 中没有 Raw 类型，如 Java 中的 List 对应于 Kotlin 就是 List<*>。

与 Java 一样，Kotlin 在运行时不保留泛型，也就是对象不携带传递到它们的构造器中的类型参数的实际类型，即 ArrayList<Integer>()和 ArrayList<Character>()

是不能区分的。这使得执行 is 检查不可能照顾到泛型，Kotlin 只允许 is 检查星投影的泛型类型。

```
if(aisList<Int>)//错误：无法检查它是否真的是一个 Int 列表
if(aisList<*>)//OK：不保证列表的内容
```

7．SAM 转换

就像 Java 8 一样，Kotlin 支持 SAM 转换，这意味着 Kotlin 函数字面值可以被自动转换成只有一个非默认方法的 Java 接口的实现，只要这个方法的参数类型能够与这个 Kotlin 函数的参数类型相匹配就行。

【例 6.4】首先使用 Java 创建一个 SAMInJava 类，然后通过 Kotlin 调用 Java 中的接口。

Java 代码：

```
packagejqiang.Mutual.Java;
import java.util.ArrayList;
public class SAMInJava{
    private ArrayList<Runnable>runnables=newArrayList<Runnable>();
    public void addTask(Runnablerunnable){
        runnables.add(runnable);
System.out.println("add:"+runnable+",size"+runnables.size());
    }
    Public void removeTask(Runnablerunnable){
        runnables.remove(runnable);
System.out.println("remove:"+runnable+"size"+runnables.size());
    }
}
```

Kotlin 代码：

```
packagejqiang.Mutual.Kotlin
importjqiang.Mutual.Java.SAMInJava
funmain(args:Array<String>){
    varsamJava=SAMInJava()
    vallamba={
        print("hello")
    }
    samJava.addTask(lamba)
    samJava.removeTask(lamba)
}
```

运行结果如下：

```
    add:jqiang.Mutual.Kotlin.SamKt$sam$Runnable$bef91c64@63947c6bsiz
e1
    remove:jqiang.Mutual.Kotlin.SamKt$sam$Runnable$bef91c64@2b193f2d
size1
```

如果 Java 类有多个接受函数式接口的方法，那么可以通过使用将 Lambda 表达式转换为特定的 SAM 类型的适配器函数来选择需要调用的方法。这些适配器函数也会按需由编译器生成。

```
    vallamba={
        print("hello")
    }
    samJava.addTask(lamba)
```

SAM 转换只适用于接口，而不适用于抽象类，即使这些抽象类只有一个抽象方法。此功能只适用于 Java 互操作；因为 Kotlin 具有合适的函数类型，所以不需要将函数自动转换为 Kotlin 接口的实现，因此不受支持。

6.1.2　Java 调用 Kotlin

在 Java 中可以轻松地调用 Kotlin 代码。

1．属性

Kotlin 属性会被编译成以下 Java 元素：

- getter 方法，其名称通过加前缀 get 得到；
- setter 方法，其名称通过加前缀 set 得到（只适用于 var 属性）；
- 私有字段，与属性名称相同（仅适用于具有幕后字段的属性）。

【例 6.5】将 Kotlin 变量编译成 Java 中的变量声明。

Kotlin 部分代码：

```
    varfirstName:String
```

Java 部分代码：

```
    privateStringfirstName;
    publicStringgetFirstName(){
        returnfirstName;
    }
    publicvoidsetFirstName(StringfirstName){
```

```
        this.firstName=firstName;
    }
```

如果属性名称是以 is 开头的，则使用不同的名称映射规则：getter 的名称与属性名称相同，并且 setter 的名称是通过将 is 替换成 set 获得的。例如，对于属性 isOpen，其 getter 会称作 isOpen()，而其 setter 会称作 setOpen()。这一规则适用于任何类型的属性，并不仅限于 Boolean。

2．包级函数

在 jqiang.Mutual.Kotlin 包内的 example.kt 文件中声明的所有函数和属性，包括扩展函数，都被编译成一个名为 jqiang.Mutual.Kotlin.ExampleKt 的 Java 类的静态方法。

【例 6.6】包级函数调用。

Kotlin 部分代码：

```
packagejqiang.Mutual.Kotlin
funbar(){
    println("这只是一个 bar 方法")
}
```

Java 部分代码：

```
packagejqiang.Mutual.Java;
publicclassexample{
    publicstaticvoidmain(String[]args){
        jqiang.Mutual.Kotlin.ExampleKt.bar();
    }
}
```

可以使用@JvmName 注解修改所生成的 Java 类的类名：

```
    @file:JvmName("example")
packagejqiang.Mutual.Kotlin
```

那么 Java 调用时就需要修改类名：

```
jqiang.Mutual.Kotlin.example.bar();
```

在多个文件中生成相同的 Java 类名（包名相同并且类名相同或者有相同的@JvmName 注解）通常是错误的。然而，编译器能够生成一个单一的 Java 外观类，它具有指定的名称且包含来自于所有文件中具有该名称的所有声明。要生成这样的外观，请在所有的相关文件中使用@JvmMultifileClass 注解。

```
@file:JvmName("example")
@file:JvmMultifileClass
packagejqiang.Mutual.Kotlin
```

3．实例字段

如果需要在 Java 中将 Kotlin 属性作为字段暴露，那么就需要使用@JvmField 注解对其进行标注。该字段将具有与底层属性相同的可见性。如果一个属性有幕后字段（Backing Field）、非私有的、没有 open/override 或者 const 修饰符，并且不是被委托的属性，那么可以使用@JvmField 注解该属性。

4．静态方法

Kotlin 将包级函数表示为静态方法。如果对这些函数使用@JvmStatic 进行标注，那么 Kotlin 还可以为在命名对象或伴生对象中定义的函数生成静态方法。如果使用该注解，那么编译器既会在相应对象的类中生成静态方法，也会在对象自身中生成实例方法。例如：

```
classC{
    companionobject{
        @JvmStaticfunfoo(){}
        funbar(){}
    }
}
```

现在，foo()在 Java 中是静态的，而 bar()不是静态的。

```
C.foo();//没问题
C.bar();//错误：不是一个静态方法
C.Companion.foo();//保留实例方法
C.Companion.bar();//唯一的工作方式
```

对于命名对象也同样：

```
objectObj{
    @JvmStaticfunfoo(){}
    funbar(){}
}
```

在 Java 中：

```
Obj.foo();//没问题
Obj.bar();//错误
Obj.INSTANCE.bar();//没问题，通过单例实例调用
Obj.INSTANCE.foo();// 也没问题
```

@JvmStatic 注解也可以被应用于对象或伴生对象的属性上，使其 getter 和 setter 方法在该对象或包含该伴生对象的类中是静态成员。

5．可见性

Kotlin 的可见性以下列方式映射到 Java。

（1）private 成员被编译成 private 成员。

（2）private 的顶层声明被编译成包级局部声明。

（3）protected 依然保持 protected（注意，Java 允许访问同一个包中其他类的受保护成员，而 Kotlin 则不允许，所以 Java 类会访问更广泛的代码）。

（4）internal 声明会成为 Java 中的 public。internal 类的成员会通过名字修饰，使其更难以在 Java 中被意外使用到，并且根据 Kotlin 规则使其允许重载相同签名的成员而互不可见。

（5）public 依然保持 public。

6．空安全性

当从 Java 中调用 Kotlin 函数时，没有任何方法可以阻止 Kotlin 中的空值传入。Kotlin 在 JVM 虚拟机中运行时会检查所有的公共函数，可以检查非空值，这时候就可以通过 NullPointerException 得到 Java 中的非空值代码。

7．型变的泛型

当 Kotlin 使用了声明处型变时，可以通过两种方式从 Java 代码中看到它们的用法。假设有以下类和两个使用它的函数：

```
class Box<out T>(val value: T)
interface Base
class Derived : Base
fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

将这两个函数转换成 Java 代码：

```
Box<Derived> boxDerived(Derived value) { … }
Base unboxBase(Box<Base> box) { … }
```

在 Kotlin 中可以这样写：unboxBase(boxDerived("s"))，但是在 Java 中是行不通的，因为在 Java 中 Box 类在其泛型参数 T 上是不型变的，于是 Box<Derived>

并不是 Box<Base>的子类。要使其在 Java 中工作，需要按以下方式定义 unboxBase：

```
Base unboxBase(Box<? extends Base> box) { … }
```

这里使用 Java 的通配符类型（? extends Base）通过使用处型变来模拟声明处型变，因为在 Java 中只能这样。

当它作为参数出现时，为了让 Kotlin 的 API 在 Java 中工作，对于协变定义的 Box 生成 Box<Super>作为 Box<?extendsSuper>（或者对于逆变定义的 Foo 生成 Foo<?superBar>）。当它是一个返回值时，则不生成通配符；否则，Java 客户端必须处理它们（并且它违反了常用的 Java 编码风格）。因此，将示例中的对应函数实际上翻译如下：

```
// 作为返回类型——没有通配符
Box<Derived> boxDerived(Derived value) { … }
// 作为参数——有通配符
Base unboxBase(Box<? extends Base> box) { … }
```

当参数类型是 final 时，生成通配符通常没有意义，所以无论在什么地方 Box<String>始终转换为 Box<String>。

如果在默认不生成通配符的地方需要通配符，则可以使用@JvmWildcard 注解。

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> =
Box(value) // 将被转换成
// Box<? extends Derived> boxDerived(Derived value) { … }
```

另外，如果根本不需要默认的通配符转换，则可以使用@JvmSuppressWildcards 注解。

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>):
Base = box.value
// 会翻译成
// Base unboxBase(Box<Base> box) { … }
```

@JvmSuppressWildcards 不仅可应用于单个类型参数，还可应用于整个声明（如函数或类），从而抑制其中的所有通配符。

6.2 Kotlin 与 JavaScript 互操作

6.2.1 Kotlin 调用 JavaScript

Kotlin 已被设计为能够与 Java 平台轻松互操作，它将 Java 类视为 Kotlin 类，并且 Java 也将 Kotlin 类视为 Java 类。但是，JavaScript 是一种动态类型语言，这意味着它不会在编译期检查类型。可以通过动态类型在 Kotlin 中自由地与 JavaScript 交流，但是如果想要利用 Kotlin 类型系统的全部威力，则可以为 JavaScript 库创建 Kotlin 头文件。

1．内联 JavaScript

可以使用 js("…")函数将一些 JavaScript 代码嵌入到 Kotlin 代码中，js 的参数必须是字符串常量。实现代码如下：

```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
```

2．external 修饰符

要告诉 Kotlin 某个声明是用纯 JavaScript 编写的，则应该用 external 修饰符来标注。当编译器看到这样的声明时，就假定相应类、函数或属性的实现由开发人员提供，因此不会尝试从声明中生成任何 JavaScript 代码。这意味着应该省略 external 声明内容的代码体。

```
external fun alert(message: Any?): Unit
    external class Node {
    val firstChild: Node
    fun append(child: Node): Node
    fun removeChild(child: Node): Node
}
external val window: Window
```

嵌套的声明会继承 external 修饰符，即在 Node 类中，在成员函数和属性之前并不放置 external。external 修饰符只允许在包级声明中使用，不能声明一个非 external 类的 external 成员。

3．声明类的（静态）成员

在 JavaScript 中，可以在原型或者类本身中定义成员。

```
function MyClass() { }
MyClass.sharedMember = function() { /* 实现 */ };
MyClass.prototype.ownMember = function() { /* 实现 */ };
```

在 Kotlin 中没有这样的语法。然而，在 Kotlin 中有伴生（Companion）对象。Kotlin 以特殊的方式处理 external 类的伴生对象，而不是期望一个对象，它会使伴随对象的成员成为该类的成员。要描述来自上例中的 MyClass，可以这样写：

```
external class MyClass {
    companion object {
        fun sharedMember()
    }
    fun ownMember()
}
```

4．扩展 JavaScript 类

扩展 JavaScript 类的一般格式如下：

```
external open class HTMLElement : Element() { /* 成员 */ }
```

虽然可以自定义 JavaScript 类，但还是有一些限制的。

（1）当一个外部基类的函数被签名重载时，不能在派生类中覆盖它。

（2）不能覆盖一个使用默认参数的函数。

5．external 接口

JavaScript 没有接口的概念，当函数期望其参数支持 foo()和 bar()函数时，只需传递实际具有这些方法的对象即可。对于静态类型的 Kotlin，可以使用接口来表达这一点。实现代码如下：

```
external interface HasFooAndBar {
    fun foo()
    fun bar()
}
external fun myFunction(p: HasFooAndBar)
```

外部接口描述设置对象，实现代码如下：

```
external interface JQueryAjaxSettings {
    var async: Boolean
```

```
    var cache: Boolean
    var complete: (JQueryXHR, String) -> Unit
}
fun JQueryAjaxSettings(): JQueryAjaxSettings = js("{}")

external class JQuery {
    companion object {
        fun get(settings: JQueryAjaxSettings): JQueryXHR
    }
}

fun sendQuery() {
    JQuery.get(JQueryAjaxSettings().apply {
    complete = { (xhr, data) ->
    window.alert("Request complete")
    }
    })
}
```

对外部接口的使用限制如下：

（1）外部接口不能在 is 检查的右侧使用。

（2）as 转换为外部接口总是成功的（在编译时产生警告）。

（3）外部接口不能作为具体化的类型参数传递。

（4）外部接口不能用在类的字面值表达式（即 I::class）中。

6.2.2 JavaScript 调用 Kotlin

Kotlin 编译器生成正常的 JavaScript 类，可以在 JavaScript 代码中自由地使用其函数和属性。

1．独立的 JavaScript 隔离声明

为了防止损坏全局对象，Kotlin 创建了一个包含当前模块中所有 Kotlin 声明的对象。所以，如果把模块命名为 myModule，那么所有的声明都可以通过 myModule 对象在 JavaScript 中使用。

```
fun foo() = "Hello"
// JavaScript 调用
alert(myModule.foo());
```

这不适用于将 Kotlin 模块编译为 JavaScript 模块时（关于这一点的详细信息，

请参见 JavaScript 模块）。在这种情况下，不会有一个包装对象，而是将声明作为相应类型的 JavaScript 模块对外暴露。例如，对于 CommonJS 的场景，应该这样写：

```
alert(require('myModule').foo())
```

2．包结构

Kotlin 将其包结构暴露给 JavaScript，因此除非在根包中定义声明，否则必须在 JavaScript 中使用完整限定的名称。例如：

```
package my.qualified.packagename
fun foo() = "Hello"
// JavaScript 调用
alert(myModule.my.qualified.packagename.foo());
```

3．@JsName 注解

在某些情况下（如为了支持重载），Kotlin 编译器会修饰（Mangle）在 JavaScript 代码中生成的函数和属性的名称。要控制所生成的名称，可以使用@JsName 注解：

```
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }
    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
// JavaScript 调用：
var person = new kjs.Person("Dmitry"); // 引用模块“kjs”
person.hello(); // 输出“Hello Dmitry!”
person.helloWithGreeting("Servus"); // 输出“Servus Dmitry!”
```

如果没有使用@JsName 注解，那么相应函数的名称会包含从函数签名计算而来的后缀，如 hello_61zpoe$。

Kotlin 编译器不会对 external 声明应用这种修饰，因此不必对其使用@JsName。值得注意的是从外部类继承的非外部类，在这种情况下，任何被覆盖的函数都不会被修饰。

@JsName 的参数需要的是一个常量字符串字面值，该字面值是一个有效的标识符。任何尝试将非标识符字符串传递给@JsName 时，编译器都会报错。

4．在 JavaScript 中表示 Kotlin 类型

（1）除了 kotlin.Long，Kotlin 的数字类型映射到 JavaScript Number。

（2）kotlin.Char 映射到 JavaScript Number 来表示字符代码。

（3）Kotlin 在运行时无法区分数字类型（kotlin.Long 除外）。

（4）Kotlin 保留了 kotlin.Int、kotlin.Byte、kotlin.Short、kotlin.Char 和 kotlin.Long 的溢出语义。

（5）在 JavaScript 中没有 64 位整数，所以 kotlin.Long 没有映射到任何 JavaScript 对象，它是由一个 Kotlin 类模拟的。

（6）kotlin.String 映射到 JavaScript String。

（7）kotlin.Any 映射到 JavaScript Object（即 newObject()、{}等）。

（8）kotlin.Array 映射到 JavaScript Array。

（9）Kotlin 集合（即 List、Set、Map 等）没有映射到任何特定的 JavaScript 类型。

（10）kotlin.Throwable 映射到 JavaScript Error。

（11）Kotlin 在 JavaScript 中保留了对象的惰性初始化。

（12）Kotlin 不会在 JavaScript 中实现顶层属性的惰性初始化。

6.3 本章小结

本章主要介绍了 Kotlin 的互操作——Kotlin 与 Java、JavaScript 互相调用。通过对本章内容的学习，我们知道了 Kotlin 的强大性，无论是 Java 还是 Javascript 都可以作为服务端来编程；而且 Kotlin 被谷歌认可作为 Android 官方开发语言，所以 Kotlin 未来不可限量。希望读者一定要深入学习 Kotlin 知识，将来成为一个全栈开发工程师。

第 7 章

电子拍卖系统

本章将介绍一个 Kotlin 项目实战案例——电子拍卖系统。

本章主要内容：

- Android 应用与 PHP 程序整合开发
- 基于 HttpClient、JSON 的数据交换整合方式
- JSON 数据与 PHP 程序整合
- 使用 PHP 开发 Android 接口
- 开发 Android 客户端

本章主要介绍一个 Android 应用：电子拍卖系统。这个应用不是一个普通的 Android 项目，而是一个 Android+PHP 整合的应用。这是一个 B/C 结构的电子拍卖系统，对于 PC 用户而言，可以通过浏览器访问该系统；对于 Android 用户而言，可以选择安装 Android 客户端程序，这样就可以通过手机来使用该拍卖系统了。

与一般书籍中所介绍的项目有所不同，本应用服务器采用 PHP 技术——在技术实现上更加依赖国内流行的 ThinkPHP 框架，该框架是目前国内性能卓越并且功能丰富的轻量级 PHP 框架，具有高度扩展性的分层结构。Android 客户端通过网络与服务端的控制器组件进行交互，整个应用具有良好的代码规范。

本应用兼容手机和平板电脑两种设备，因此需要为手机和平板电脑设计良好的用户界面。该应用程序采用大量的 Fragment，而 Fragment 代表可复用的“Activity 片段”，因此兼容两种屏幕的界面复用了共同的 Fragment。

7.1 系统功能简介和架构设计

本章介绍的应用不是一个简单的单机小应用，而是一个 Android 与传统服务器数据结合的应用，在这个应用中，服务端程序依然保持了良好的应用架构，而客户端使用 Android 程序充当。

7.1.1 系统功能介绍

本章介绍的电子拍卖系统是从实际的电子商务平台抽离出来的，只是取其中最核心的功能实现作为实战案例应用，向读者展示一种良好的程序架构。

电子拍卖系统其实就是一个小型的电子商务平台，只要在应用服务器中部署该系统，全球的客户就可以在该系统中发布想出售的商品，也可以对拍卖中的商品进行竞价。整个过程无须人工干预，系统自动完成。

如果该系统提供了支付接口，那么就可以通过该支付接口进行支付，实现买家自动付款，一旦付款成功，卖家就可以将买家所购买的商品通过物流中心发送到买家手中。可见，这个电子拍卖系统是一种开放式、成本低的系统，大量工作都无须人工干预，系统会自动完成。当然，该系统只是一个例子，因此不提供支付接口，只是模拟用户添加拍卖商品、参与拍卖等基本行为，拍卖结束后系统判定商品是否被最高竞价者获取。

本系统的服务端是采用 ThinkPHP 内核完成的一个电子拍卖项目。Android 客户端只负责与服务端的控制器进行交互，Android 应用 Apache HttpClient 向服务端的控制器发送请求并获取服务器响应信息，这样即可实现 Android 与电子拍卖系统之间的通信。

本系统要求用户在参与拍卖之前，必须先登录系统。在系统中根据 Session 中的值来判断用户是否登录，如果没有登录就进入登录界面；如果已经登录，则继续参与竞拍。

通过本系统可以查询拍卖商品、添加拍卖商品和商品种类、进行竞价处理，以及发送邮件通知用户所参与的竞价商品。

注册会员可以添加拍卖商品和拍卖商品种类，但是必须先登录系统。

注册用户可以浏览当前拍卖的商品，以及流拍的商品。

注册用户可以参与商品竞价，参与竞价系统通过邮件通知用户。

7.1.2 系统架构设计

本系统的服务端采用的是 PHP 的分层结构，分为模型（Model）、视图（View）和控制器（Controller）。分层体系将业务规则、数据访问等工作放在模型中处理，客户端不直接与数据库进行交互，而是通过控制器调用模型中的方法。

- 模型层是应用程序的核心部分，它可以是实体对象或一种业务逻辑。之所以称之为模型，是因为它在应用程序中有更好的重用性和扩展性。

- 视图层提供应用程序与用户之间进行交互的界面，在 MVC 中，这一层不包含任何业务逻辑，仅提供一种与用户交互的视图。
- 控制器层用于对程序中的请求进行控制，其作用相当于国家的宏观调控，它可以选择调用视图或模型。

本系统使用 MySQL 数据库存储数据。本系统的服务端应用总体架构示意图如图 7-1 所示。

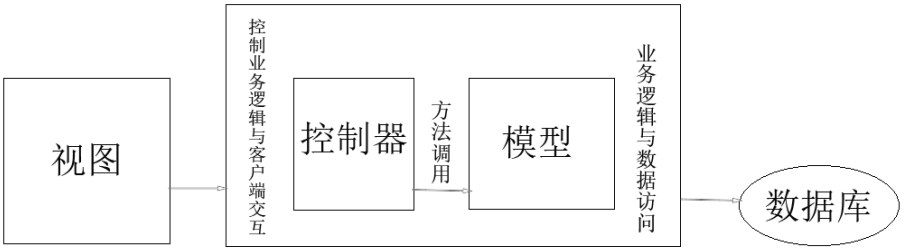


图 7-1

作为客户端的 Android 应用可以通过网络与服务器交互，它会通过 Apache HttpClient 向服务端的控制器发送请求，并获取服务器响应。服务器响应采用 JSON 数据格式，可以有效地进行数据交互。

Android 应用向服务器发送请求，此处的入口是对应控制器中的方法，这样非常便于快速开发。

如图 7-2 所示为 Android 客户端与服务器的整体架构示意图。

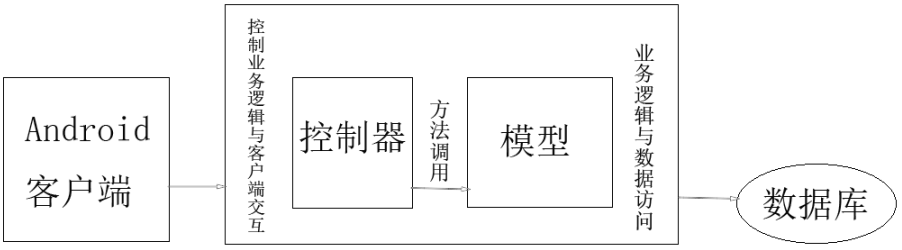


图 7-2

细心的读者可能已经发现，图 7-1 和图 7-2 十分相似，如果有传统的 PC 页面，那么只需要在原来的控制器的基础上进行简单的修改，就可以提供 Android 客户端数据接口；如果没有，则可以按照自己的想法重新编写控制器，作为 PC 页面的控制器。

7.2 JSON 简介

Android 客户端与服务器通信需要一种合适的数据交换格式，本系统采用 JSON 作为数据交换格式。

JSON（JavaScript Object Notation，JavaScript 对象标记）是一种轻量级的数据交换格式，它基于 ECMAScript（W3C 制定的 JavaScript 规范）的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言，易于人阅读和编写，同时也易于机器解析和生成，并有效提高了网络传输效率。

在 JavaScript 语言中，一切都是对象。因此，它所支持的任何类型都可以通过 JSON 来表示，如字符串、数字、对象、数组等。对象和数组是比较特殊且常用的两种类型。

对象：在 JavaScript 中是使用花括号“{}”包裹起来的内容，其数据结构为{key1：value1,key2：value2,...}这样的键值对结构。在面向对象语言中，key 为对象的属性，value 为对应的值。键名可以使用整数和字符串来表示；值可以是任意类型的。

数组：在 JavaScript 中是使用方括号“[]”包裹起来的内容，其数据结构为["java","javascript","vb",...]这样的索引结构。在 JavaScript 中，数组是一种比较特殊的数据类型，它也可以像对象那样使用键值对，但还是索引使用得多。同样，值可以是任意类型的。

7.2.1 使用 PHP 创建 JSON 数据对象

通过 PHP 生成 JSON 数据非常简单，使用 json_encode()函数即可。但是需要明确一点，json_encode()函数中的参数编码必须为 UTF-8，否则将返回 null。

在生成接口数据时，数据需要满足标准格式。

（1）状态码（用来标识服务器的状态，这时客户端就能识别这种状态）。

（2）信息提示，如登录成功、数据返回失败等。

（3）数据（封装的数据）。

在 PHP 中创建 JSON 对象，需要与数组进行结合，生成能与 Android 客户端

进行完美交互的数据类型。我们可以通过如下方式来创建数组。

```
<?php
$arr = array(
    'code' => '200',
    'msg' => '成功',
    'data' => array(
        'id' => '1',
        'name' => 'Kotlin',
        )
    );
$json_string = json_encode($arr);
echo $json_string;
```

电子拍卖系统采用的是轻量级 PHP 开发框架 ThinkPHP，所以该系统会使用 ThinkPHP 中内置的方法来编写对应的业务逻辑，以及对客户端进行响应。

7.2.2　接口交互工具类

本系统通过 ThinkPHP 与 Android 客户端进行数据交互。为了给 Android 客户端提供良好的数据响应信息，基于 ThinkPHP 框架，封装一个可以和 Android 客户端进行交互的工具类。

- show()：向客户端提供的响应信息默认为 JSON 数据。
- json()：向客户端提供的响应信息为 JSON 数据。
- xmlEncode()：向客户端提供的响应信息为 XML 数据。

该工具类的代码如下：

```
<?php
class Response {
    const JSON = "json";
    /**
    * 按综合方式输出通信数据
    * @param integer $code 状态码
    * @param string $msg 提示信息
    * @param array $data 数据
    * @param string $type 数据类型
    * return string
    */
    public static function show($code, $msg = '', $data = array(),
```

```
$type = self::JSON) {
          if(!is_numeric($code)) {
             return '';
          }

          $type   =   isset($_GET['format'])   ?   $_GET['format']   :
self::JSON;

          $result = array(
             'code' => $code,
             'msg' => $msg,
             'data' => $data,
          );

          if($type == 'json') {
             self::json($code, $msg, $data);
             exit;
          } elseif($type == 'array') {
             var_dump($result);
          } elseif($type == 'xml') {
             self::xmlEncode($code, $msg, $data);
             exit;
          } else {
             // TODO
          }
       }
       /**
       * 按 JSON 方式输出通信数据
       * @param integer $code 状态码
       * @param string $msg 提示信息
       * @param array $data 数据
       * return string
       */
       public static function json($code, $msg = '', $data = array()) {

          if(!is_numeric($code)) {
             return '';
          }

          $result = array(
             'code' => $code,
             'msg' => $msg,
             'data' => $data
```

```
            );

            echo json_encode($result);
            exit;
        }

        /**
        * 按 XML 方式输出通信数据
        * @param integer $code 状态码
        * @param string $msg 提示信息
        * @param array $data 数据
        * return string
        */
        public static function xmlEncode($code, $msg, $data = array()) {
            if(!is_numeric($code)) {
                return '';
            }

            $result = array(
                'code' => $code,
                'msg' => $msg,
                'data' => $data,
            );

            header("Content-Type:text/xml");
            $xml = "<?xml version='1.0' encoding='UTF-8'?>\n";
            $xml .= "<root>\n";
            $xml .= self::xmlToEncode($result);
            $xml .= "</root>";
            echo $xml;
        }

        public static function xmlToEncode($data) {

            $xml = $attr = "";
            foreach($data as $key => $value) {
                if(is_numeric($key)) {
                    $attr = " id='{$key}'";
                    $key = "item";
                }
                $xml .= "<{$key}{$attr}>";
                $xml .= is_array($value) ? self::xmlToEncode($value) :
$value;
```

```
            $xml .= "</{$key}>\n";
        }
        return $xml;
    }
}
```

上面在 Response 类中定义了 show()、json()和 xmlEncode()三个方法，通过这三个方法服务器向客户端提供响应数据，并返回数据到 Android 客户端。在 ToApi 中提供这三个方法之后，接下来在服务器应用中只要调用其中的 show()方法即可实现与 Android 客户端的通信。

7.3 发送请求的工具类

本系统通过 Apache HttpClient 与远程服务器通信。为了简化 HttpCHent 的用法，本系统定义了一个工具类对 HttpClient 进行封装，该工具类定义了如下两个方法来发送请求。

- getRequest()：发送 GET 请求。
- postRequest()：发送 POST 请求。

该工具类的代码如下：

```
object HttpUtil {
    // 创建 HttpClient 对象
    var httpClient: HttpClient = DefaultHttpClient()
    val BASE_URL = "http://127.0.0.1/api.php/"
    /**

     * @param url 发送请求的 URL
     * @return 服务器响应字符串
     * @throws Exception
     */
    @Throws(Exception::class)
    fun getRequest(url: String): String {
        val task = FutureTask(
                Callable<String> {
                    // 创建 HttpGet 对象
                    val get = HttpGet(url)
                    // 发送 GET 请求
                    val httpResponse = httpClient.execute(get)
```

```
                    // 如果服务器成功返回响应信息
                    if (httpResponse.getStatusLine()
                            .getStatusCode() === 200) {
                        // 获取服务器响应字符串
                        val result = EntityUtils
                                .toString(httpResponse.getEntity())
                        return@Callable result
                    }
                    null
                })
            Thread(task).start()
            return task.get()
        }

        /**
         * @param url 发送请求的 URL
         * @param params 请求参数
         * @return 服务器响应字符串
         * @throws Exception
         */
        @Throws(Exception::class)
        fun postRequest(url: String, rawParams: Map<String, String>):
String {
            val task = FutureTask(
                    Callable<String> {
                        // 创建 HttpPost 对象。
                        val post = HttpPost(url)
                        // 如果传递的参数个数比较多的话，则可以对所传递的参数进行
封装
                        val params = ArrayList<NameValuePair>()
                        for (key in rawParams.keys) {
                            //封装请求参数
                            params.add(BasicNameValuePair(key,
rawParams[key]))
                        }
                        // 设置请求参数
                        post.setEntity(UrlEncodedFormEntity(
                                params, "gbk"))
                        // 发送 POST 请求
                        val httpResponse = httpClient.execute(post)
                        // 如果服务器成功返回响应信息
                        if (httpResponse.getStatusLine()
                                .getStatusCode() === 200) {
```

```
                    // 获取服务器响应字符串
                    val result = EntityUtils
                            .toString(httpResponse.getEntity())
                    return@Callable result
                }
                null
            })
        Thread(task).start()
        return task.get()
    }
}
```

上面在 HttpUtil 中定义了 getRuquest()和 postRequest()两个方法，这两个方法用于向服务器发送请求，并返回服务器的响应信息。在 HttpUtil 中提供这两个方法之后，接下来在 Android 应用中只要调用这两个方法即可实现与服务器的通信。

7.4 用户登录

在使用该电子拍卖系统之前，用户必须先登录系统。用户在 Android 应用中输入用户名、密码，单击“登录”按钮，程序即可通过 HttpUtil 向服务器发送请求，通过服务器来验证用户所输入的用户名、密码是否正确。

7.4.1 处理登录的 LoginController

在 LoginController 控制器中提供了处理用户登录的 do_login()方法，该方法的作用如下：

- 获取请求参数。
- 业务逻辑处理。
- 根据处理结果生成输出信息。

由于本系统的服务端采用的是通过 ThinkPHP 框架编写的能与 Android 客户端交互的数据接口，因此程序需要借助 ThinkPHP 中内置的方法来编写用户登录业务逻辑，并将结果提供给客户端。

下面是用于处理用户登录的 LoginController 控制器中的 do_login()方法的代码。

```
    public function do_login(){
```

```
        $info=M('User')->where(['username'=>I('username'),
'userpass'=>I('userpass')])->find();
        if ($info) {
            session('userId',$info['user_id']);
                $this->ajaxReturn($info['user_id']);
        }else{
            $this->ajaxReturn("-1");
        }
    }
```

7.4.2　用户登录客户端

在 Android 客户端的用户登录界面中有两个文本框，用于接收用户输入的用户名和密码信息。

下面就是用户登录界面布局的 XML 文档。

```
    <?xml version="1.0" encoding="utf-8"?>
    <TableLayout   xmlns:android="http://schemas.android.com/apk/res/
android"
        android:layout_width="300dp"
        android:layout_height="match_parent"
        android:layout_gravity="center_horizontal"
        android:stretchColumns="1">
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="fitCenter"
        android:src="@drawable/logo"
        android:contentDescription="@string/hello"/>
    <TextView
        android:text="@string/welcome"
        android:id="@+id/TextView"
        android:textSize="@dimen/label_font_size"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:padding="@dimen/title_padding"/>
    <!-- 输入用户名的行 -->
    <TableRow>
    <TextView
        android:text="@string/user_name"
        android:textSize="@dimen/label_font_size"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/userEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text"/>
    </TableRow>
    <!-- 输入密码的行 -->
    <TableRow>
    <TextView
        android:text="@string/user_pass"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:text=""
        android:id="@+id/pwdEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPassword"/>
    </TableRow>
    <!-- 定义登录、取消按钮的行 -->
    <LinearLayout android:orientation="horizontal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center">
    <Button
        android:id="@+id/bnLogin"
        android:text="@string/login"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button
        android:id="@+id/bnCancel"
        android:text="@string/cancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    </LinearLayout>
    </TableLayout>
```

在登录界面中输入用户名、密码后，单击“登录”按钮，会激发登录处理程序，也就是通过 HttpUtil 向服务器发送请求。用户登录的 Activity 代码如下：

```
    class Login : Activity() {
```

```
        // 定义登录界面中的两个文本框
        internal var etName: EditText
        internal var etPass: EditText
        // 定义登录界面中的两个按钮
        internal var bnLogin: Button
        internal var bnCancel: Button
        public override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.login)
            // 获取登录界面中的两个文本框
            etName = findViewById(R.id.userEditText) as EditText
            etPass = findViewById(R.id.pwdEditText) as EditText
            // 获取登录界面中的两个按钮
            bnLogin = findViewById(R.id.bnLogin) as Button
            bnCancel = findViewById(R.id.bnCancel) as Button
            // 为 bnCancal 按钮的单击事件绑定事件监听器
            bnCancel.setOnClickListener(HomeListener(this))
            bnLogin.setOnClickListener {
                // 执行输入校验
                if (validate())
                // ①
                {
                    // 如果登录成功
                    if (loginPro())
                    // ②
                    {
                        // 启动 Main Activity
                        val intent = Intent(this@Login,
AuctionClientActivity::class.java)
                        startActivity(intent)
                        // 结束该 Activity
                        finish()
                    } else {
                        DialogUtil.showDialog(this@Login, "用户名或者密码错
误，请重新输入！", false)
                    }
                }
            }
        }

        private fun loginPro(): Boolean {
            // 获取用户输入的用户名、密码
            val username = etName.text.toString()
```

```
            val pwd = etPass.text.toString()
            val jsonObj: JSONObject
            try {
                jsonObj = query(username, pwd)
                // 如果 userId 大于 0
                if (jsonObj.getInt("userId") > 0) {
                    return true
                }
            } catch (e: Exception) {
                DialogUtil.showDialog(this, "服务器响应异常，请稍后再试！",
false)
                e.printStackTrace()
            }

            return false
        }

        // 对用户输入的用户名、密码进行校验
        private fun validate(): Boolean {
            val username = etName.text.toString().trim { it <= ' ' }
            if (username == "") {
                DialogUtil.showDialog(this, "用户名是必填项！", false)
                return false
            }
            val pwd = etPass.text.toString().trim { it <= ' ' }
            if (pwd == "") {
                DialogUtil.showDialog(this, "用户密码是必填项！", false)
                return false
            }
            return true
        }

        // 定义发送请求的方法
        @Throws(Exception::class)
        private fun query(username: String, password: String): JSONObject
{
            // 使用 Map 封装请求参数
            val map = HashMap<String, String>()
            map.put("username", username)
            map.put("userpass", password)
            // 定义发送请求的 URL
            val url = HttpUtil.BASE_URL + "/Login/do_login"
            // 发送请求
```

```
        return JSONObject(HttpUtil.postRequest(url, map))
    }
}
```

上面 Activity 中的 query()方法用于向指定 URL 发送消求，并将服务器响应封装成 JSONObject。

上面程序为登录按钮的单击事件绑定了事件监听器，当用户单击“登录”按钮时，程序先执行输入校验，执行登录处理程序，然后调用 LoginPro 来处理用户的登录请求。

如果用户输入的用户名、密码不正确，系统将会调用 DialogUtil 来显示对话框，提示登录失败。由于本系统经常需要显示各种对话框，因此程序专门把它定义成一个独立的类。DialogUtil 类的代码如下：

```
object DialogUtil {
    // 定义一个显示消息的对话框
    fun showDialog(ctx: Context, msg: String, goHome: Boolean) {
        // 创建一个 AlertDialog.Builder 对象
        val builder = AlertDialog.Builder(ctx)
                .setMessage(msg).setCancelable(false)
        if (goHome) {
            builder.setPositiveButton("确定") { dialog, which ->
                val i = Intent(ctx, AuctionClientActivity::class.java)
                i.flags = Intent.FLAG_ACTIVITY_CLEAR_TOP
                ctx.startActivity(i)
            }
        } else {
            builder.setPositiveButton("确定", null)
        }
        builder.create().show()
    }

    // 定义一个显示指定组件的对话框
    fun showDialog(ctx: Context, view: View) {
        AlertDialog.Builder(ctx)
                .setView(view).setCancelable(false)
                .setPositiveButton("确定", null)
                .create()
                .show()
    }
}
```

如果登录成功，系统将启动 AuctionClientActivity，这个 Activity 相当于系统

的主界面，用户可以通过该界面提供的 ListView 进入各种功能中。

AuctionClientActivity 的界面布局文件有两个：为普通手机屏幕提供的界面布局文件和为平板电脑屏幕提供的界面布局文件。下面先看为普通手机屏幕提供的界面布局文件。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <!-- 添加一个 Fragment -->
    <fragment
        android:name="org.crazyit.auction.client.
AuctionListFragment"
        android:id="@+id/auction_list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

上面程序界面中包含的主要组件就是 AuctionListFragment，该 Fragment 显示一个 ListView，在该 ListView 中每一个列表项代表一个系统功能。

下面再看为平板电脑屏幕提供的界面布局文件。

```
<?xml version="1.0" encoding="utf-8"?>
<!-- 定义一个水平排列的 LinearLayout，并指定使用中等分隔条 -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    android:divider="?android:attr/dividerHorizontal"
    android:showDividers="middle">
    <!-- 添加一个 Fragment -->
    <fragment
        android:name="org.crazyit.auction.client.
AuctionListFragment"
        android:id="@+id/auction_list"
        android:layout_width="0dp"
        android:layout_height="match_parent"
```

```
            android:layout_weight="1" />
        <!-- 添加一个 FrameLayout 容器 -->
        <FrameLayout
            android:id="@+id/auction_detail_container"
            android:layout_width="0dp"
            android:paddingLeft="10dp"
            android:layout_height="match_parent"
            android:layout_weight="3" />
    </LinearLayout>
```

上面程序界面中同样包含一个 AuctionListFragment。除此之外，该程序界面中还包含一个 FrameLayout 容器，该容器主要负责装载功能的 Fragment 组件。

由于在 layout 和 layout-sw480dp 目录下分别包含了 activity_main.xml 界面布局文件，因此 AuctionClientActivity 会根据屏幕尺寸自动加载不同目录下的界面布局文件。除此之外，该 Activity 必须根据界面布局来进行处理——如果程序加载 layout 目录下的 activity_main.xml 界面布局文件，当用户单击功能项时，系统将会启动相应的 Activity 来显示功能；如果程序加载 layout-sw480dp 目录下的 activity_main.xml 界面布局文件，当用户单击功能项时，系统将会使用 FrameLayout 加载相应功能的 Fragment 组件。

```
    class AuctionClientActivity : Activity(), Callbacks {
        // 定义一个旗标，用于标识该应用是否支持大屏幕
        private var mTwoPane: Boolean = false

        public override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            // 指定加载 R.layout.activity_main 对应的界面布局文件
            // 但实际上该应用会根据屏幕分辨率加载不同的界面布局文件
            setContentView(R.layout.activity_main)
            // 如果所加载的界面布局文件中包含 id 为 auction_detail_container 的
组件
            if (findViewById(R.id.auction_detail_container) != null) {
                mTwoPane = true
                (fragmentManager
                        .findFragmentById(R.id.auction_list) as
AuctionListFragment)
                        .setActivateOnItemClick(true)
            }
        }

        override fun onItemSelected(id: Int?, bundle: Bundle) {
```

```
            if (mTwoPane) {
                var fragment: Fragment? = null
                when (id as Int) {
                    // 查看竞得商品
                    0 -> {
                        // 创建 ViewItemFragment
                        fragment = ViewItemFragment()
                        // 创建 Bundle，准备向 Fragment 传入参数
                        val arguments = Bundle()
                        arguments.putString("action", "/Item/viewSucc ")
                        // 向 Fragment 传入参数
                        fragment.arguments = arguments
                    }
                    // 浏览流拍商品
                    1 -> {
                        // 创建 ViewItemFragment
                        fragment = ViewItemFragment()
                        // 创建 Bundle，准备向 Fragment 传入参数
                        val arguments2 = Bundle()
                        arguments2.putString("action", "/Item/viewFail ")
                        // 向 Fragment 传入参数
                        fragment.arguments = arguments2
                    }
                    // 管理商品种类
                    2 ->
                        // 创建 ManageKindFragment
                        fragment = ManageKindFragment()
                    // 管理商品
                    3 ->
                        // 创建 ManageItemFragment
                        fragment = ManageItemFragment()
                    // 浏览拍卖商品（选择商品种类）
                    4 ->
                        // 创建 ChooseKindFragment
                        fragment = ChooseKindFragment()
                    // 查看自己的竞拍商品
                    5 ->
                        // 创建 ViewBidFragment
                        fragment = ViewBidFragment()
                    ManageItemFragment.ADD_ITEM -> fragment =
AddItemFragment()
                    ManageKindFragment.ADD_KIND -> fragment =
AddKindFragment()
```

```
                ChooseKindFragment.CHOOSE_ITEM -> {
                    fragment = ChooseItemFragment()
                    val args = Bundle()
                    args.putLong("kindId", bundle.getLong("kindId"))
                    fragment.arguments = args
                }
                ChooseItemFragment.ADD_BID -> {
                    fragment = AddBidFragment()
                    val args2 = Bundle()
                    args2.putInt("itemId", bundle.getInt("itemId"))
                    fragment.arguments = args2
                }
            }
            // 使用 fragment 替换 auction_detail_container 容器当前显示的
Fragment
            fragmentManager.beginTransaction()
                    .replace(R.id.auction_detail_container,fragment)
                    .addToBackStack(null).commit()
        } else {
            var intent: Intent? = null
            when (id as Int) {
                // 查看竞得商品
                0 -> {
                    // 启动 ViewItem Activity
                    intent = Intent(this, ViewItem::class.java)
                    // action 属性为请求的 Servlet 地址
                    intent.putExtra("action", "/Item/viewSucc ")
                    startActivity(intent)
                }
                // 浏览流拍商品
                1 -> {
                    // 启动 ViewItem Activity
                    intent = Intent(this, ViewItem::class.java)
                    // action 属性为请求的 Servlet 的 URL
                    intent.putExtra("action", "/Item/viewFail")
                    startActivity(intent)
                }
                // 管理商品种类
                2 -> {
                    // 启动 ManageKind Activity
                    intent = Intent(this, ManageKind::class.java)
                    startActivity(intent)
```

```
                }
                // 管理商品
                3 -> {
                    // 启动 ManageItem Activity
                    intent = Intent(this, ManageItem::class.java)
                    startActivity(intent)
                }
                // 浏览拍卖商品（选择商品种类）
                4 -> {
                    // 启动 ChooseKind Activity
                    intent = Intent(this, ChooseKind::class.java)
                    startActivity(intent)
                }
                // 查看自己的竞拍商品
                5 -> {
                    // 启动 ViewBid Activity
                    intent = Intent(this, ViewBid::class.java)
                    startActivity(intent)
                }
            }
        }
    }
}
```

从上面代码可以看出，如果在手机上运行，AuctionClientActivity 主要提供一个 ListView，当用户单击不同的列表项时，程序将会启动不同的 Activity。

用户单击“主菜单”（其实是 ListView）的任一个列表项，系统将会启动对应的 Activity，从而允许用户执行相应的操作。

如果在平板电脑上运行，AuctionClientActivity 则提供一个 ListView 和一个 FrameLayout 容器，当用户单击不同的列表项时，程序将使用 FrameLayout 加载相应功能的 Fragment。

用户单击“主菜单”（其实是 ListView）的任一个列表项，系统将会使用 FrameLayout 加载相应功能的 Fragment。

7.5 查看流拍商品

当用户单击“浏览流拍商品”时，程序将使用 FrameLayout 加载相应功能的 Fragment，并显示系统中的流拍商品。

7.5.1　查看流拍商品的 ItemController

查看流拍商品的 ItemController，也是通过 ThinkPHP 中内置的方法对业务处理生成 ItemView()方法的，该方法用于获取流拍商品。

该 ItemController 控制器中的 ItemView()方法的代码如下：

```
   public function ItemView(){
      $list=M('Item')->field('item_id,item_name,item_remark,
item_desc,addtime,endtime,init_price,max_price')->select();

      $State=M('State')->find();
      $User=M('User')->find();
      $kind=M('kind')->find();
      foreach ($list as $key => $value) {
         $list[$key]['state']=$State['state_name'];
         $list[$key]['owner']=$User['username'];
         $list[$key]['kind']=$kind['kind_name'];
      }
      $this->ajaxReturn($list);
   }
```

在上面程序中，通过调用 ThinkPHP 框架的核心方法编写获取流派商品的数据列表。同时通过 ajaxReturn()方法返回 JSON 数据，并且将响应信息输出到客户端。

如果直接通过浏览器向/Item/ItemView 发送请求，将会看到如图 7-3 所示的输出信息。

图 7-3

图 7-3 显示的字符串就是典型的 JSON 格式，Android 客户端只要调用 JSON 数据即可。

7.5.2 查看流拍商品客户端

下面是查看流拍商品的程序界面布局代码。

```
    <?xml version="1.0" encoding="utf-8"?>
    <LinearLayout  xmlns:android="http://schemas.android.com/apk/res/
android"
        android:orientation="vertical"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    <LinearLayout
        android:orientation="horizontal"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/sub_title_margin">
    <TextView
        android:id="@+id/view_titile"
        android:text="@string/view_succ"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <!-- 定义返回按钮 -->
    <Button
        android:id="@+id/bn_home"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/label_font_size"
        android:background="@drawable/home"/>
    </LinearLayout>
    <!-- 查看商品列表的 ListView -->
    <ListView
        android:id="@+id/succList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
    </LinearLayout>
```

上面页面是通过 XML 语言编写的，用于显示多个商品。查看流拍商品的 Fragment 代码如下：

```
    class ViewItemFragment : Fragment() {
        internal var bnHome: Button
```

```
    internal var succList: ListView
    internal var viewTitle: TextView
    override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?, savedInstanceState: Bundle): View? {
        val rootView = inflater.inflate(R.layout.view_item,
container, false)
        // 获取界面中的返回按钮
        bnHome = rootView.findViewById(R.id.bn_home) as Button
        succList = rootView.findViewById(R.id.succList) as ListView
        viewTitle = rootView.findViewById(R.id.view_titile) as
TextView
        // 为返回按钮的单击事件绑定事件监听器
        bnHome.setOnClickListener(HomeListener(activity))
        val action = arguments.getString("action")
        // 定义发送请求的 URL
        val url = HttpUtil.BASE_URL + action!!
        // 如果是查看流拍商品，则修改标题
        if (action == "/Item/ItemView") {
            viewTitle.setText(R.string.view_fail)
        }
        try {
            // 向指定 URL 发送请求，并把服务器响应转换成 JSONArray 对象
            val jsonArray = JSONArray(HttpUtil
                    .getRequest(url)) // 将 JSONArray 包装成 Adapter
            val adapter = JSONArrayAdapter(activity, jsonArray,
"name", true)
            succList.adapter = adapter
        } catch (e: Exception) {
            DialogUtil.showDialog(activity, "服务器响应异常，请稍后再试!
", false)
            e.printStackTrace()
        }

        succList.onItemClickListener = OnItemClickListener { parent,
view, position, id ->
            // 查看指定商品的详细情况
            viewItemDetail(position) }
        return rootView
    }

    private fun viewItemDetail(position: Int) {
        // 加载 detail.xml 界面布局代表的视图
        val detailView = activity.layoutInflater
```

```
                .inflate(R.layout.detail, null)
        // 获取 detail.xml 界面布局中的文本框
        val itemName = detailView
                .findViewById(R.id.itemName) as TextView
        val itemKind = detailView
                .findViewById(R.id.itemKind) as TextView
        val maxPrice = detailView
                .findViewById(R.id.maxPrice) as TextView
        val itemRemark = detailView
                .findViewById(R.id.itemRemark) as TextView
        // 获取被单击的列表项
        val jsonObj = succList.adapter.getItem(
                position) as JSONObject
        try {
            // 通过文本框显示商品详情
            itemName.text = jsonObj.getString("item_name")
            itemKind.text = jsonObj.getString("kind")
            maxPrice.text = jsonObj.getString("max_Price")
            itemRemark.text = jsonObj.getString("item_desc")
        } catch (e: JSONException) {
            e.printStackTrace()
        }

        DialogUtil.showDialog(activity, detailView)
    }
}
```

在程序中只需向 URL 发送请求，并把服务器响应包装成 JSONArray 对象，就可实现 Android 客户端与服务器的交互。JSONArray 对象的本质就是一个数组，它提供了如下常用的方法。

- length()：返回该 JSON 数组的长度。
- optJSONObiject(int index)：获取索引处的 JSONObject 对象。
- optJSONArray(int index)：获取索引处的 JSONArray 对象。
- optXxx(int index)：获取指定索引处的数组元素。

对于开发者来说，单纯获取 JSONArray 对象之后，完全可以把它当成数组来处理。

本程序提供了一个 JSONArrayAdapter 类，它的本质就是一个 Adapter，该 Adapter 用于对 JSON 数组进行包装，并作为 ListView 的内容 Adapter。JSONArrayAdapter 类的代码如下：

```
    class JSONArrayAdapter(private val ctx: Context, // 定义需要包装的
JSONArray 对象
            private val jsonArray: JSONArray, // 定义列表项显示 JSONObject
对象的哪个属性
            private val property: String, private val hasIcon: Boolean) :
BaseAdapter() {

        override fun getCount(): Int {
            return jsonArray.length()
        }

        override fun getItem(position: Int): Any {
            return jsonArray.optJSONObject(position)
        }

        override fun getItemId(position: Int): Long {
            try {
                // 返回商品的 id
                return (getItem(position) as JSONObject).getInt("id").
toLong()
            } catch (e: JSONException) {
                e.printStackTrace()
            }

            return 0
        }

        override fun getView(position: Int, convertView: View, parent:
ViewGroup): View {
            // 定义一个线性布局管理器
            val linear = LinearLayout(ctx)
            // 设置为水平的线性布局管理器
            linear.orientation = 0
            // 创建一个 ImageView
            val iv = ImageView(ctx)
            iv.setPadding(10, 0, 20, 0)
            iv.setImageResource(R.drawable.item)
            // 将图片添加到 LinearLayout 中
            linear.addView(iv)
            // 创建一个 TextView
            val tv = TextView(ctx)
            try {
                // 获取 JSONArray 数组元素的 property 属性
```

```
            val itemName = (getItem(position) as JSONObject)
                    .getString(property)
            // 设置 TextView 所显示的内容
            tv.text = itemName
        } catch (e: JSONException) {
            e.printStackTrace()
        }

        tv.textSize = 20f
        if (hasIcon) {
            // 将 TextView 添加到 LinearLayout 中
            linear.addView(tv)
            return linear
        } else {
            return tv
        }
    }
}
```

上面的 Adapter 负责为 ListView 提供列表项。ListView 的列表项包括如下两种情况。

（1）当 ListView 的列表项有图标时，该 Adapter 所提供的每一个列表项就是一个 LinearLayout（里面包含一个图标和一个 TextView）。

（2）当 ListView 的列表项没有图标时，该 Adapter 所提供的每一个列表项就是一个普通的 TextView。

上面的 JSONArrayAdapter 可以多次使用，因此程序把它单独定义成一个 Adapter 类，这样可以多次使用该类 Adapter 来封装 JSONArray 对象。

在平板电脑上查看流拍商品的列表，只能看到流拍商品的商品名，如果用户希望看到该商品的详情，则可以单击该商品对应的列表项，此时程序将会触发 ViewItemFragment 中的 viewltemDetail(position)方法，如 ViewItemFragment 类中的代码所示。

viewItemDetail(position)方法会使用对话框加载系统中的 detail.xml 界面布局文件，并显示该流拍商品的详情。当用户单击流拍商品的列表项时，系统将会显示商品的详情。

上面的 ViewItemFragment 除用于显示流拍商品之外，也可用于查看竞得商品，因为这两个功能的前端实现基本上是一致的，只是 Android 需要调用的服务器方法不同而已。

如果在手机上使用该系统，则还需要使用一个 Activity 来“盛装”该 Fragment。由于系统中有大量的 Fragment 需要使用 Activity 来显示，因此本系统专门开发了一个 FragmentActivity，用于显示一个 Fragment。该 Activity 的代码如下：

```
public abstract class FragmentActivity extends Activity
{
    private static final int ROOT_CONTAINER_ID = 0x90001;
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        LinearLayout layout = new LinearLayout(this);
        setContentView(layout);
        layout.setId(ROOT_CONTAINER_ID);
        getFragmentManager().beginTransaction()
            .replace(ROOT_CONTAINER_ID , getFragment())
            .commit();
    }
    protected abstract Fragment getFragment();
}
```

上面代码定义了一个抽象方法，该抽象方法将返回一个 Fragment，而该 Activity 的作用就是加载、显示该 Fragment。

FragmentActivity 用于被其他 Activity 继承，继承它的 Activity 只需重写 getFragment()方法即可。ViewItem Activity 的代码如下：

```
class ViewItem : FragmentActivity() {
    // 重写 getFragment()方法，该 Activity 显示该方法返回的 Fragment
    override fun getFragment(): Fragment {
        val fragment = ViewItemFragment()
        val arguments = Bundle()
        arguments.putString("action",intent.getStringExtra("action"))
        fragment.arguments = arguments
        return fragment
    }
}
```

从上面代码不难看出，ViewItem Activity 只是用于显示 ViewItemFragment，在显示 Fragment 之前向该 Fragment 传入参数。

7.6 管理商品种类

管理商品种类包括浏览系统中的商品种类、添加商品种类两大主要功能。Android 应用主要充当与用户交互的客户端：显示系统中的商品种类；添加商品种类时，提供文本框供用户输入种类名称、种类描述等必要信息。

7.6.1 浏览商品种类的 KindController

浏览商品种类的 KindController 控制器，通过 ThinkPHP 中内置的方法自定义一个 KindView()方法来查询商品种类。该 KindView()方法的代码如下：

```
public function KindView(){
    $list=M('Kind')->select();
    $this->ajaxReturn($list);
}
```

该查询返回 JSON 数据，把 JSON 数据作为服务器响应。

使用浏览器直接浏览 KindController 控制器中的 KindView 方法，可以看到如图 7-4 所示的 JSON 数据

图 7-4

7.6.2 查看商品种类

在 Android 客户端查看商品种类，需要先通过 HttpUtil 向服务器发送请求，并把服务器响应字符串转换成 JSONArray 对象，然后使用 Adapter 包装 JSONArray 对象，再使用 ListView 进行显示。

查看商品种类的界面布局代码如下：

```
    <?xml version="1.0" encoding="utf-8"?>
    <LinearLayout  xmlns:android="http://schemas.android.com/apk/res/
android"
        android:orientation="vertical"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    <LinearLayout
        android:orientation="horizontal"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/sub_title_margin">
    <LinearLayout
        android:orientation="vertical"
        android:gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
    <TextView
        android:text="@string/manage_kind"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <!-- 添加种类的按钮 -->
    <Button
        android:id="@+id/bnAdd"
        android:layout_width="85dp"
        android:layout_height="30dp"
        android:background="@drawable/add_kind"/>
    </LinearLayout>
    <Button
        android:id="@+id/bn_home"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/label_font_size"
        android:background="@drawable/home"/>
    </LinearLayout>
    <!-- 显示种类列表的 ListView -->
    <ListView
        android:id="@+id/kindList"
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"/>
    </LinearLayout>
```

在上面的界面布局中定义了一个 ListView，用来显示系统中的商品种类。程序只需把服务器返回的商品种类包装成 Adapter，并使用该 ListView 来显示所有的商品种类即可。下面是显示商品种类的 Fragment 代码。

```
    class ManageKindFragment : Fragment() {
        internal var bnHome: Button
        internal var bnAdd: Button
        internal var kindList: ListView
        internal var mCallbacks: Callbacks? = null
        override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?, savedInstanceState: Bundle): View? {
            val   rootView   =   inflater.inflate(R.layout.manage_kind,
container, false)
            // 获取界面布局中的两个按钮
            bnHome = rootView.findViewById(R.id.bn_home) as Button
            bnAdd = rootView.findViewById(R.id.bnAdd) as Button
            kindList = rootView.findViewById(R.id.kindList) as ListView
            // 为返回按钮的单击事件绑定事件监听器
            bnHome.setOnClickListener(HomeListener(activity))
            // 为添加种类按钮的单击事件绑定事件监听器
            bnAdd.setOnClickListener {
                // 当添加种类按钮被单击时，
                // 调用该 Fragment 所在 Activity 的 onItemSelected 方法
                mCallbacks!!.onItemSelected(ADD_KIND, null)
            }
            // 定义发送请求的 URL
            val url = HttpUtil.BASE_URL + "/Kind/KindView"
            try {
                // 向指定 URL 发送请求，并把响应包装成 JSONArray 对象
                val jsonArray = JSONArray(
                        HttpUtil.getRequest(url))
                // 把 JSONArray 对象包装成 Adapter
                kindList.adapter = KindArrayAdapter(jsonArray, activity)
            } catch (e: Exception) {
                DialogUtil.showDialog(activity, "服务器响应异常，请稍后再试!
", false)
                e.printStackTrace()
            }

            return rootView
```

```
        }

        // 当该 Fragment 被添加、显示到 Activity 时，回调该方法
        override fun onAttach(activity: Activity) {
            super.onAttach(activity)
            // 如果 Activity 没有实现 Callbacks 接口，则抛出异常
            if (activity !is Callbacks) {
                throw IllegalStateException(
                    "ManageKindFragment 所在的 Activity 必须实现 Callbacks 接
口!")
            }
            // 把该 Activity 当成 Callbacks 对象
            mCallbacks = activity
        }

        // 当该 Fragment 从它所属的 Activity 中被删除时，回调该方法
        override fun onDetach() {
            super.onDetach()
            // 将 mCallbacks 赋值为 null
            mCallbacks = null
        }

        companion object {
            val ADD_KIND = 0x1007
        }
    }
```

在上面程序中，onCreateView()方法中的粗体字代码是实现向服务器发送请求，把服务器响应转换成 JSONArray 对象，并使用 ListView 显示商品种类的核心代码。由于商品种类包含的信息量并不大，只有种类名称和种类描述，因此程序使用 KindArrayAdapter 来包装 JSONArray 对象，KindArrayAdapter 所提供的列表项既包括了种类名称，又包括了种类描述。下面是该 Adapter 类的代码。

```
    class KindArrayAdapter(// 需要包装的 JSONArray
        private val kindArray: JSONArray, private val ctx: Context) :
BaseAdapter() {
        override fun getCount(): Int {
            // 返回 ListView 所显示的列表项的数量
            return kindArray.length()
        }

        override fun getItem(position: Int): Any {
            // 获取指定列表项所包装的 JSONObject
```

```
            return kindArray.optJSONObject(position)
        }

        override fun getItemId(position: Int): Long {
            try {
                return (getItem(position) as JSONObject).getInt("id").
toLong()
            } catch (e: JSONException) {
                e.printStackTrace()
            }

            return -1
        }

        override fun getView(position: Int, convertView: View,
                             parent: ViewGroup): View {
            // 定义一个线性布局管理器
            val container = LinearLayout(ctx)
            // 设置为水平的线性布局管理器
            container.orientation = 1
            // 定义一个线性布局管理器
            val linear = LinearLayout(ctx)
            // 设置为水平的线性布局管理器
            linear.orientation = 0
            // 创建一个 ImageView
            val iv = ImageView(ctx)
            iv.setPadding(10, 0, 20, 0)
            iv.setImageResource(R.drawable.item)
            // 将图片添加到 LinearLayout 中
            linear.addView(iv)
            // 创建一个 TextView
            val tv = TextView(ctx)
            try {
                // 获取 JSONArray 数组元素的 kindName 属性
                val kindName = (getItem(position) as JSONObject)
                        .getString("kindName")
                // 设置 TextView 所显示的内容
                tv.text = kindName
            } catch (e: JSONException) {
                e.printStackTrace()
            }

            tv.textSize = 20f
```

```
            // 将 TextView 添加到 LinearLayout 中
            linear.addView(tv)
            container.addView(linear)
            // 定义一个文本框来显示种类描述
            val descView = TextView(ctx)
            descView.setPadding(30, 0, 0, 0)
            try {
                // 获取 JSONArray 数组元素的 kindDesc 属性
                val kindDesc = (getItem(position) as JSONObject)
                        .getString("kindDesc")
                descView.text = kindDesc
            } catch (e: JSONException) {
                e.printStackTrace()
            }

            descView.textSize = 16f
            container.addView(descView)
            return container
        }
    }
```

从上面程序中可以看出，该 Adapter 提供的每一个列表项不仅包括种类名称，而且包括种类描述，这样用户查看种类时一目了然。

为了在手机上运行该应用，程序提供了一个 ManageKindActivity 来包装、显示该 Fragment。ManageKind 的代码如下：

```
    class ManageKind : FragmentActivity(), Callbacks {
        public override fun getFragment(): Fragment {
            return ManageKindFragment()
        }

        override fun onItemSelected(id: Int?, bundle: Bundle) {
            // 当用户单击 ManageKindFragment 中的添加种类按钮时，启动 AddKind
Activity
            val i = Intent(this, AddKind::class.java)
            startActivity(i)
        }
    }
```

在商品界面中有一个“添加种类”按钮，当用户单击该按钮时，Fragment 将会回调它所在 Activity 中的 onItemSelected()方法。对于在平板电脑上运行的 Activity，onItemSelected()方法将会让 FrameLayout 加载 AddKindFragment；对于

在手机上运行的 Activity，onItemSelected()方法将会启动 AddKindActivity（该 Activity 仅仅是包装、显示 AddIKindFragment）。

7.6.3 添加商品种类的 KindController

添加商品种类的 KindController 控制器，也是通过 ThinkPHP 中内置的方法来编写添加商品种类的 addKind()方法的。该 KindController 控制器中的 addKind()方法的代码如下：

```
    public function addKind(){
        $data=I('post.');
        $info=M('Kind')->data($data)->add();
        if ($info) {
            $this->ajaxReturn("恭喜您，种类添加成功");
        }else{
            $this->ajaxReturn("种类添加失败");
        }
    }
```

上面 KindController 中的用于添加商品种类的 addKind()方法运行后，将直接返回添加结果的字符串数据，因此 Android 客户端只能直接显示添加结果。

7.6.4 添加商品种类

在添加商品种类的界面中有两个输入框，用于接收用户输入的种类名称和种类描述，并提供添加、取消两个按钮。该界面布局比较简单，此处不再给出界面布局代码。

添加商品种类的 Fragment 代码如下：

```
    class AddKindFragment : Fragment() {
        // 定义界面中的两个文本框
        internal var kindName: EditText
        internal var kindDesc: EditText
        // 定义界面中的两个按钮
        internal var bnAdd: Button
        internal var bnCancel: Button
        override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?, savedInstanceState: Bundle): View? {
            val rootView = inflater.inflate(R.layout.add_kind, container,
false)
```

```
            // 获取界面中的两个文本框
            kindName = rootView.findViewById(R.id.kindName) as EditText
            kindDesc = rootView.findViewById(R.id.kindDesc) as EditText
            // 获取界面中的两个按钮
            bnAdd = rootView.findViewById(R.id.bnAdd) as Button
            bnCancel = rootView.findViewById(R.id.bnCancel) as Button
            // 为取消按钮的单击事件绑定事件监听器
            bnCancel.setOnClickListener(HomeListener(activity))
            bnAdd.setOnClickListener {
                // 输入校验
                if (validate()) {
                    // 获取用户输入的种类名称、种类描述
                    val name = kindName.text.toString()
                    val desc = kindDesc.text.toString()
                    try {
                        // 添加商品种类
                        val result = addKind(name, desc)
                        // 使用对话框来显示添加结果
                        DialogUtil.showDialog(activity, result, true)
                    } catch (e: Exception) {
                        DialogUtil.showDialog(activity, "服务器响应异常，请
稍后再试！", false)
                        e.printStackTrace()
                    }

                }
            }
            return rootView
        }

        // 对用户输入的种类名称进行校验
        private fun validate(): Boolean {
            val name = kindName.text.toString().trim { it <= ' ' }
            if (name == "") {
                DialogUtil.showDialog(activity, "种类名称是必填项！",
false)
                return false
            }
            return true
        }

        @Throws(Exception::class)
        private fun addKind(name: String, desc: String): String {
```

```
        // 使用 Map 封装请求参数
        val map = HashMap<String, String>()
        map.put("kindName", name)
        map.put("kindDesc", desc)
        // 定义发送请求的 URL
        val url = HttpUtil.BASE_URL + "/Kind/add"
        // 发送请求
        return HttpUtil.postRequest(url, map)
    }
}
```

上面的 Fragment 先对用户输入的种类名称、种类描述进行校验，然后调用 addKind()方法来添加商品种类，该方法利用 HttpUtil 发送 POST 请求完成添加。

为了兼容手机屏幕，同时定义 Activity 来加载、显示该 Fragment。AddKind Activity 的代码如下：

```
class AddKind : FragmentActivity() {
    public override fun getFragment(): Fragment {
        return AddKindFragment()
    }
}
```

正如上面代码所示，AddKind Activity 什么都没干，仅仅是加载、显示 AddKindFragment。在手机上添加商品种类，如果添加成功，将看到如图 7-5 所示的对话框。

图 7-5

7.7 管理拍卖商品

管理拍卖商品包括浏览自己的拍卖商品、添加拍卖商品两大主要功能。Android 应用主要充当与用户交互的客户端：显示当前用户的拍卖商品；添加拍卖商品时，提供文本框供用户输入商品名称、商品描述等必要信息。

7.7.1　查看自己的拍卖商品的 ItemController

在 ItemController 控制器中，用于查看自己的拍卖商品的 ItemownerView()方法也是通过 ThinkPHP 中内置的方法查询表数据的。该 ItemownerView()方法的代码如下：

```
    public function ItemownerView(){
    $list=M('Item')->field('item_id,item_name,item_remark,item_desc,
addtime,endtime,init_price,max_price')->where('owner_id='.
session('user_id'))->select();
        $State=M('State')->find();
        $User=M('User')->find();
        $kind=M('kind')->find();
        foreach ($list as $key => $value) {
            $list[$key]['state']=$State['state_name'];
            $list[$key]['owner']=$User['username'];
            $list[$key]['kind']=$kind['kind_name'];
        }
        $this->ajaxReturn($list);
    }
```

上面 ItemController 中的 ItemownerView()方法是通过 ThinkPHP 框架内置的方法进行业务处理，然后将拍卖商品以 JSON 对象提供给 Android 客户端进行交互的。

使用浏览器直接浏览上面的 ItemownerView 方法，将会看到如图 7-6 所示的 JSON 数据。

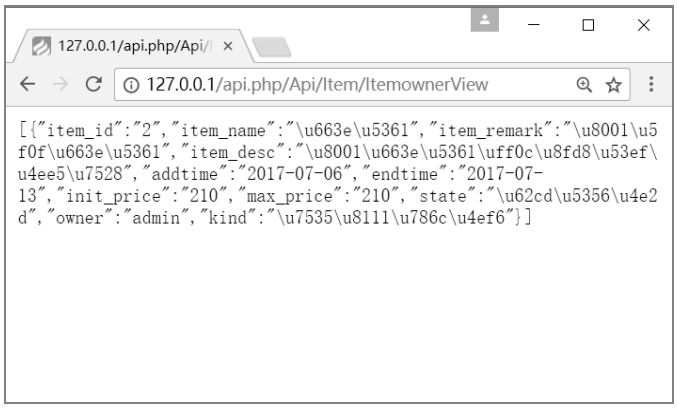


图 7-6

当服务器返回如图 7-6 所示的 JSON 数据后，Android 客户端就可以把它转换成 JSONArray 对象，并从中获取详细的商品信息。

7.7.2 查看自己的拍卖商品

在查看拍卖商品界面中使用 ListView 来显示商品列表，该界面布局的代码如下：

```
    <?xml version="1.0" encoding="utf-8"?>
    <LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    <LinearLayout
        android:orientation="horizontal"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/sub_title_margin">
    <LinearLayout
        android:orientation="vertical"
        android:gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingLeft="12dp">
    <TextView
        android:text="@string/manage_item"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <!-- 添加商品的按钮 -->
    <Button
        android:id="@+id/bnAdd"
        android:layout_width="85dp"
        android:layout_height="30dp"
        android:background="@drawable/add_item"/>
    </LinearLayout>
    <Button
        android:id="@+id/bn_home"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:layout_marginLeft="@dimen/label_font_size"
        android:background="@drawable/home"/>
    </LinearLayout>
    <!-- 显示商品列表的 ListView -->
    <ListView
        android:id="@+id/itemList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
    </LinearLayout>
```

在上面的界面布局中定义了一个 ListView 来显示当前用户的拍卖商品。除此之外，该界面中还包含了几个按钮，其中一个按钮用于启动添加商品的用户界面。

管理商品的 Fragment 通过 HttpUtil 向服务器发送请求，并把服务器响应转换成 JSONArray 对象，然后把 JSONArray 对象包装成 Adapter，最后使用 ListView 来显示这些商品。管理商品的 Fragment 的代码如下：

```
    class ManageItemFragment : Fragment() {
        internal var bnHome: Button
        internal var bnAdd: Button
        internal var itemList: ListView
        internal var mCallbacks: Callbacks? = null

        override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?, savedInstanceState: Bundle): View? {
            val rootView = inflater.inflate(R.layout.manage_item,
container, false)
            bnHome = rootView.findViewById(R.id.bn_home) as Button
            bnAdd = rootView.findViewById(R.id.bnAdd) as Button
            itemList = rootView.findViewById(R.id.itemList) as ListView
            // 为返回按钮的单击事件绑定事件监听器
            bnHome.setOnClickListener(HomeListener(activity))
            bnAdd.setOnClickListener
{ mCallbacks!!.onItemSelected(ADD_ITEM, null) }
            // 定义发送请求的 URL
            val url = HttpUtil.BASE_URL + "/Item/ItemownerView"
            try {
                // 向指定 URL 发送请求
                val jsonArray = JSONArray(HttpUtil.getRequest(url))
                // 将服务器响应包装成 Adapter
                val adapter = JSONArrayAdapter(activity, jsonArray,
"name", true)
                itemList.adapter = adapter
```

```
            } catch (e: Exception) {
                DialogUtil.showDialog(activity, "服务器响应异常，请稍后再试!
", false)
                e.printStackTrace()
            }

            itemList.onItemClickListener = OnItemClickListener { parent,
view, position, id ->
                viewItemInBid(position)  // ①
            }
            return rootView
        }

        // 当该 Fragment 被添加、显示到 Activity 时，回调该方法
        override fun onAttach(activity: Activity) {
            super.onAttach(activity)
            // 如果 Activity 没有实现 Callbacks 接口，则抛出异常
            if (activity !is Callbacks) {
                throw IllegalStateException(
                        "ManagerItemFragment 所在的 Activity 必须实现
Callbacks 接口!")
            }
            // 把该 Activity 当成 Callbacks 对象
            mCallbacks = activity
        }

        // 当该 Fragment 从它所属的 Activity 中被删除时，回调该方法
        override fun onDetach() {
            super.onDetach()
            // 将 mCallbacks 赋值为 null
            mCallbacks = null
        }

        private fun viewItemInBid(position: Int) {
            // 加载 detail_in_bid.xml 界面布局文件代表的视图
            val detailView = activity.layoutInflater
                    .inflate(R.layout.detail_in_bid, null)
            // 获取 detail_in_bid.xml 界面中的文本框
            val itemName = detailView
                    .findViewById(R.id.itemName) as TextView
            val itemKind = detailView
                    .findViewById(R.id.itemKind) as TextView
            val maxPrice = detailView
```

```
                .findViewById(R.id.maxPrice) as TextView
        val initPrice = detailView
                .findViewById(R.id.initPrice) as TextView
        val endTime = detailView
                .findViewById(R.id.endTime) as TextView
        val itemRemark = detailView
                .findViewById(R.id.itemRemark) as TextView
        // 获取被单击列表项所包装的 JSONObject
        val jsonObj = itemList.adapter.getItem(
                position) as JSONObject
        try {
            // 通过文本框显示商品详情
            itemName.text = jsonObj.getString("item_name")
            itemKind.text = jsonObj.getString("kind")
            maxPrice.text = jsonObj.getString("maxPrice")
            itemRemark.text = jsonObj.getString("item_desc")
            initPrice.text = jsonObj.getString("initPrice")
            endTime.text = jsonObj.getString("endTime")
        } catch (e: JSONException) {
            e.printStackTrace()
        }

        DialogUtil.showDialog(activity, detailView)
    }

    companion object {
        val ADD_ITEM = 0x1006
    }
}
```

上面程序中的粗体字代码使用了 HttpUtil 向服务器发送请求，并使用 ListView 来显示服务器返回的商品列表。当用户在平板电脑上进入管理商品的界面时，将会看到如图 7-7 所示的界面。

图 7-7 只是列出了所拍卖商品的商品名称，如果用户需要查看该商品的详情，则可以单击代表该商品的列表项，程序将会触发 viewItemInBid(position)方法（如上面程序中的①号粗体字代码所示），该方法将会启动一个对话框，其中定义了多个文本框用来显示该商品的详情。

图 7-7

为了兼容手机屏幕，同时还定义了 Activity 来加载、显示该 Fragment。ManageItem Activity 的代码如下：

```
class ManageItem : FragmentActivity(), Callbacks {
    public override fun getFragment(): Fragment {
        return ManageItemFragment()
    }

    override fun onItemSelected(id: Int?, bundle: Bundle) {
        // 当用户单击添加按钮时，将会启动 AddItem Activity
        val i = Intent(this, AddItem::class.java)
        startActivity(i)
    }
}
```

在图 7-7 所示的界面中包含了一个“添加商品”按钮，单击该按钮可以添加拍卖商品。

7.7.3 添加拍卖商品的 ItemController

在 ItemController 控制器中定义一个 addItem()方法用于实现添加拍卖商品功能。该 addItem()方法的代码如下：

```
public function addItem(){
$data=I('post.');
$data['owner_id']=session('user_id');
    $info=M('Item')->data($data)->add();
    if ($info) {
```

```
            $this->ajaxReturn("恭喜商品添加成功");
        }else{
            $this->ajaxReturn("商品添加失败");
        }
    }
```

上面 ItemController 控制器中的用于添加拍卖商品的 addItem()方法运行后，将直接返回添加结果的字符串数据，因此 Android 客户端只能直接显示添加结果。

7.7.4　添加拍卖商品

在添加拍卖商品的界面中包含了多个文本框，用于接收商品的属性信息。添加拍卖商品的界面布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1">
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/add_item_title"
    android:textSize="@dimen/label_font_size"
    android:gravity="center"
    android:padding="@dimen/title_padding"/>
<!-- 输入商品名称的行 -->
<TableRow>
<TextView
    android:text="@string/item_name"
    android:textSize="@dimen/label_font_size"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
<EditText
    android:id="@+id/itemName"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="text"/>
</TableRow>
<!-- 输入商品描述的行 -->
<TableRow>
```

```
    <TextView
        android:text="@string/item_desc"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/itemDesc"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text"/>
    </TableRow>
    <!-- 输入商品备注的行 -->
    <TableRow>
    <TextView
        android:text="@string/remark"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/itemRemark"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text"/>
    </TableRow>
    <!-- 输入起拍价格的行 -->
    <TableRow>
    <TextView
        android:text="@string/init_price"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/initPrice"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="numberDecimal"/>
    </TableRow>
    <!-- 选择有效时间的行 -->
    <TableRow>
    <TextView
        android:text="@string/avail_time"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"/>
    <Spinner
        android:id="@+id/availTime"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:entries="@array/availTime"/>
    </TableRow>
    <!-- 选择商品种类的行 -->
    <TableRow>
    <TextView
        android:text="@string/item_kind"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Spinner
        android:id="@+id/itemKind"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
    </TableRow>
    <!-- 定义按钮的行 -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center">
    <Button
        android:id="@+id/bnAdd"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/add"/>
    <Button
        android:id="@+id/bnCancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/cancel"/>
    </LinearLayout>
    </TableLayout>
```

在上面的界面布局中包含了两个 Spinner 组件，用来定义两个列表框供用户选择有效时间和商品种类。对于选择有效时间的 Spinner 来说，程序可以直接指定一个数组作为它的列表项；但对于选择商品种类的 Spinner 来说，程序必须加载系统中的所有商品种类作为它的列表项。因此，为了在 Spinner 中加载商品种类，程序必须向/Item/viewKind 发送请求，然后把响应包装成 Adapter，再使用

Spinner 来显示商品种类列表。

在添加拍卖商品的界面中，最下面的商品种类就是从系统中加载的。当用户填写了拍卖商品的详情之后，程序将会向/Item/addItem 发送请求来添加拍卖商品。添加拍卖商品的 Fragment 代码如下：

```
    class AddItemFragment : Fragment() {
        // 定义界面中的文本框
        internal var itemName: EditText
        internal var itemDesc: EditText
        internal var itemRemark: EditText
        internal var initPrice: EditText
        internal var itemKind: Spinner
        internal var availTime: Spinner
        // 定义界面中的两个按钮
        internal var bnAdd: Button
        internal var bnCancel: Button
        override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?, savedInstanceState: Bundle): View? {
            val rootView = inflater.inflate(R.layout.add_item, container,
false)
            // 获取界面中的文本框
            itemName = rootView.findViewById(R.id.itemName) as EditText
            itemDesc = rootView.findViewById(R.id.itemDesc) as EditText
            itemRemark  =  rootView.findViewById(R.id.itemRemark)  as
EditText
            initPrice  =  rootView.findViewById(R.id.initPrice)  as
EditText
            itemKind = rootView.findViewById(R.id.itemKind) as Spinner
            availTime = rootView.findViewById(R.id.availTime) as Spinner
            // 定义发送请求的地址
            val url = HttpUtil.BASE_URL + "/Item/ItemView"
            var jsonArray: JSONArray? = null
            try {
                // 获取系统中的所有商品种类
                // 向执行 URL 发送请求，并把服务器响应包装成 JSONArray
                jsonArray = JSONArray(HttpUtil.getRequest(url))  // ①
            } catch (e1: Exception) {
                e1.printStackTrace()
            }

            // 将 JSONArray 包装成 Adapter
            val adapter = JSONArrayAdapter(
```

```
                activity, jsonArray, "kindName", false)
        // 显示商品种类列表
        itemKind.adapter = adapter
        // 获取界面中的两个按钮
        bnAdd = rootView.findViewById(R.id.bnAdd) as Button
        bnCancel = rootView.findViewById(R.id.bnCancel) as Button
        // 为取消按钮的单击事件绑定事件监听器
        bnCancel.setOnClickListener(HomeListener(activity))
    bnAdd.setOnClickListener {
        // 执行输入校验
        if (validate()) {
            // 获取用户输入的商品名称、商品描述等信息
            val name = itemName.text.toString()
            val desc = itemDesc.text.toString()
            val remark = itemRemark.text.toString()
            val price = initPrice.text.toString()
            val kind = itemKind.selectedItem as JSONObject
                var avail = availTime.selectedItemPosition
                //根据用户选择的有效时间选项，指定实际的有效时间
                when (avail) {
                    5 -> avail = 7
                    6 -> avail = 30
                    else -> avail += 1
                }
                try {
                    // 添加商品
                    val result = addItem(name, desc, remark, price,
kind.getInt("id"), avail)
                    // 显示对话框
                    DialogUtil.showDialog(activity, result, true)
                } catch (e: Exception) {
                    DialogUtil.showDialog(activity, "服务器响应异常，请
稍后再试！", false)
                    e.printStackTrace()
                }

            }
        }
        return rootView
    }

    // 对用户输入的商品名称、起拍价格进行校验
    private fun validate(): Boolean {
```

```
            val name = itemName.text.toString().trim { it <= ' ' }
            if (name == "") {
                DialogUtil.showDialog(activity, "商品名称是必填项！",
false)
                return false
            }
            val price = initPrice.text.toString().trim { it <= ' ' }
            if (price == "") {
                DialogUtil.showDialog(activity, "起拍价格是必填项！",
false)
                return false
            }
            try {
                // 尝试把起拍价格转换为浮点数
                java.lang.Double.parseDouble(price)
            } catch (e: NumberFormatException) {
                DialogUtil.showDialog(activity, "起拍价格必须是数值！",
false)
                return false
            }

            return true
        }

        @Throws(Exception::class)
        private fun addItem(name: String, desc: String, remark: String,
initPrice: String, kindId: Int, availTime: Int): String {
            // 使用 Map 封装请求参数
            val map = HashMap<String, String>()
            map.put("itemName", name)
            map.put("itemDesc", desc)
            map.put("itemRemark", remark)
            map.put("initPrice", initPrice)
            map.put("kindId", kindId.toString() + "")
            map.put("availTime", availTime.toString() + "")
            // 定义发送请求的 URL
            val url = HttpUtil.BASE_URL + "/Item/addItem"
            // 发送请求
            return HttpUtil.postRequest(url, map)
        }
    }
```

上面程序中的①号粗体字代码向/Item/viewKind 发送请求，并把服务器响应包

装成 JSONArray 对象，然后使用 Spinner 显示商品种类列表。

当填写了拍卖商品的详情之后，单击“添加”按钮将先执行输入校验，然后调用 addItem()方法来添加拍卖商品，如上面程序中的两行粗体字代码所示。如果用户添加拍卖商品成功，将会显示商品添加成功的提示信息。

为了兼容手机屏幕，同时定义了 Activity 来加载、显示该 Fragment。AddItem Activity 的代码如下：

```
class AddItem : FragmentActivity() {
    public override fun getFragment(): Fragment {
        return AddItemFragment()
    }
}
```

从上面的代码不难看出，AddItem Activity 只是用于加载并显示 AddItemFragment。

7.8　竞拍商品

用户可以通过商品种类浏览系统中的拍卖商品，找到合适的拍卖商品之后，参与该商品的竞价。

7.8.1　选择商品种类

前面介绍的/Item/ItemView 可以生成所有商品种类的响应，Android 客户端只要向该 ItemController 发送请求，即可显示商品种类列表供用户选择。在选择商品种类的界面中主要包含一个 ListView，该 ListView 列出了系统中的所有商品种类，当用户单击某个商品种类时，程序将会显示示该种类下的全部商品。

选择商品种类的 Fragment 代码如下：

```
class ChooseKindFragment : Fragment() {
    internal var mCallbacks: Callbacks? = null
    internal var bnHome: Button
    internal var kindList: ListView
    override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?, savedInstanceState: Bundle): View? {
        val rootView = inflater.inflate(R.layout.choose_kind,
container, false)
```

```
            bnHome = rootView.findViewById(R.id.bn_home) as Button
            kindList = rootView.findViewById(R.id.kindList) as ListView
            // 为返回按钮的单击事件绑定事件监听器
            bnHome.setOnClickListener(HomeListener(activity))
            // 定义发送请求的 URL
            val url = HttpUtil.BASE_URL + "/Item/ItemView"
            try {
            // 向指定 URL 发送请求，并将服务器响应包装成 JSONArray 对象
                val jsonArray = JSONArray(
                        HttpUtil.getRequest(url))  // ①
                // 使用 ListView 显示所有商品种类
                kindList.adapter = KindArrayAdapter(jsonArray, activity)
            } catch (e: Exception) {
                DialogUtil.showDialog(activity, "服务器响应异常，请稍后再试!
", false)
                e.printStackTrace()
            }

            kindList.onItemClickListener = OnItemClickListener { parent,
view, position, id ->
                val bundle = Bundle()
                bundle.putLong("kindId", id)
                mCallbacks!!.onItemSelected(CHOOSE_ITEM, bundle)
            }
            return rootView
        }

        // 当该 Fragment 被添加、显示到 Activity 时，回调该方法
        override fun onAttach(activity: Activity) {
            super.onAttach(activity)
            // 如果 Activity 没有实现 Callbacks 接口，则抛出异常
            if (activity !is Callbacks) {
                throw IllegalStateException(
                        "ManageKindFragment 所在的 Activity 必须实现
Callbacks 接口!")
            }
            // 把该 Activity 当成 Callbacks 对象
            mCallbacks = activity
        }

        // 当该 Fragment 从它所属的 Activity 中被删除时，回调该方法
```

```
        override fun onDetach() {
            super.onDetach()
            // 将 mCallbacks 赋值为 null
            mCallbacks = null
        }

        companion object {
            val CHOOSE_ITEM = 0x1008
        }
    }
```

程序中①号粗体字代码用于向指定 URL 发送请求，并将服务器响应转换成 JSONArray 对象，接下来程序只要把 JSONArray 对象包装成 Adapter，并使用 ListView 显示商品种类列表即可。

当用户单击指定的商品种类时，程序将会回调该 Fragment 所在 Activity 中的 onItemSelected()方法，并把当前商品种类作为参数传递过去，而该 Fragment 所在的 Activity 将会负责加载 Fragment，或启动对应的 Activity 来显示该种类下的全部拍卖商品。

7.8.2　根据种类浏览商品的 ItemController

在 ItemController 中定义了一个只要得到 kind_id 值就可以获取商品列表的 ItemlistView()方法。

下面就是 ItemlistView()方法的代码。

```
    public function ItemlistView(){
        $kind_id=I('kind_id');
    $list=M('Item')->field('item_id,item_name,item_remark,item_desc,
addtime,endtime,init_price,max_price')->where('kind_id='.$kind_id)->
select();
        $State=M('State')->find();
        $User=M('User')->find();
        $kind=M('kind')->find();
        foreach ($list as $key => $value) {
            $list[$key]['state']=$State['state_name'];
            $list[$key]['owner']=$User['username'];
            $list[$key]['kind']=$kind['kind_name'];
        }
```

```
        $this->ajaxReturn($list);
    }
```

使用浏览器直接访问 ItemController 控制器中的 ItemlistView 方法，将可以看到如图 7-8 所示的数据

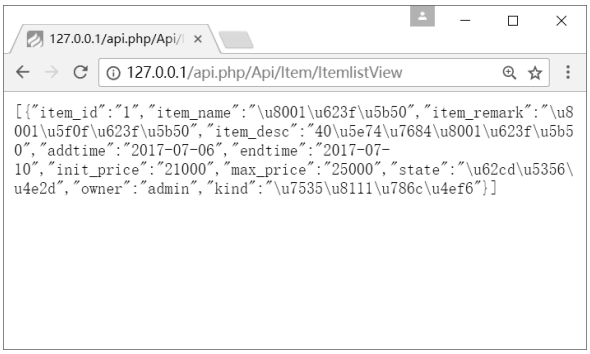


图 7-8

7.8.3 根据种类浏览商品

服务器 ItemlistView()方法生成如图 7-8 所示的 JSON 数据响应后，Android 客户端只要向该 ItemController 控制器发送请求，并对 JSON 数据进行重新赋值，再将该 JSON 数据包装成 Adapter，并使用 ListView 来显示这些商品即可。

下面是 ChooseItemFragment 的代码。

```
    class ChooseItemFragment : Fragment() {
        internal var bnHome: Button
        internal var succList: ListView
        internal var viewTitle: TextView
        internal var mCallbacks: Callbacks? = null
        // 重写该方法，该方法返回的 View 将作为 Fragment 显示的组件
        override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?, savedInstanceState: Bundle): View? {
            val   rootView   =   inflater.inflate(R.layout.view_item,
container, false)
            // 获取界面中的返回按钮
            bnHome = rootView.findViewById(R.id.bn_home) as Button
            succList = rootView.findViewById(R.id.succList) as ListView
            viewTitle   =   rootView.findViewById(R.id.view_titile)   as
TextView
```

```
            // 为返回按钮的单击事件绑定事件监听器
            bnHome.setOnClickListener(HomeListener(activity))
            val kindId = arguments.getLong("kindId")
            // 定义发送请求的 URL
            val url = HttpUtil.BASE_URL + "/Item/ItemlistView?kind_id ="
+ kindId
            try {
                // 根据种类 ID 获取该种类对应的所有商品
                val jsonArray = JSONArray(HttpUtil.getRequest(url))
                val adapter = JSONArrayAdapter(
                        activity, jsonArray, "name", true)
                // 使用 ListView 显示当前种类下的所有商品
                succList.adapter = adapter
            } catch (e1: Exception) {
                DialogUtil.showDialog(activity, "服务器响应异常，请稍后再试!
", false)
                e1.printStackTrace()
            }

            // 修改标题
            viewTitle.setText(R.string.item_list)
            succList.onItemClickListener = OnItemClickListener { parent,
view, position, id ->
                val jsonObj = succList
                        .adapter.getItem(position) as JSONObject
                val bundle = Bundle()
                try {
                    bundle.putInt("itemId", jsonObj.getInt("id"))
                } catch (e: JSONException) {
                    e.printStackTrace()
                }

                mCallbacks!!.onItemSelected(ADD_BID, bundle)
            }
            return rootView
        }

        // 当该 Fragment 被添加、显示到 Activity 时，回调该方法
        override fun onAttach(activity: Activity) {
            super.onAttach(activity)
            // 如果 Activity 没有实现 Callbacks 接口，则抛出异常
```

```
            if (activity !is Callbacks) {
                throw IllegalStateException(
                        "ManagerItemFragment 所在的 Activity 必须实现
Callbacks 接口!")
            }
            // 把该 Activity 当成 Callbacks 对象
            mCallbacks = activity
        }

        // 当该 Fragment 从它所属的 Activity 中被删除时，回调该方法
        override fun onDetach() {
            super.onDetach()
            // 将 mCallbacks 赋值为 null
            mCallbacks = null
        }

        companion object {
            val ADD_BID = 0x1009
        }
    }
```

程序中粗体字代码用于向指定 URL 发送请求，并将服务器响应转换成 JSONArray 对象，接下来程序只要把 JSONArray 对象包装成 Adapter，并使用 ListView 显示商品种类列表即可。

7.8.4 参与竞价的 ItemController

参与竞价其实就是将 Android 客户端的值，包含竞价商品的 id 与当前用户登录的 id 等信息保存在 bid 表中。因此可以通过在 ItemController 控制器中添加一个 addBid()方法来实现该功能。

addBid()方法的代码如下：

```
    // 参与竞价
    public function addBid(){
    $data['item_id']=I('post.item_id');
    $data['bid_price']=I('post.bid_price');
    $data['owner_id']=session('user_id');
        $data['bid_date']=time();
        $info=M('Bid')->data($data)->add();
        if ($info) {
```

```
            $this->ajaxReturn("恭喜竞价成功");
        }else{
            $this->ajaxReturn("竞价失败");
        }
    }
```

上面 ItemController 控制器中的用于添加竞价记录的 addBid()方法运行后，将直接返回添加结果的字符串数据，因此 Android 客户端只能直接显示添加结果。

7.8.5　参与商品竞价

当用户选择指定的商品参与竞价时，系统将会显示该商品的当前详情，例如起拍价格、当前最高竞价等，在界面下方提供了一个文本框供用户输入竞拍价格。

竞拍界面布局的代码如下：

```
    <?xml version="1.0" encoding="utf-8"?>
    <ScrollView
xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    <TableLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:stretchColumns="1">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/item_detail_title"
        android:textSize="@dimen/label_font_size"
        android:gravity="center"
        android:padding="@dimen/title_padding"/>
    <!-- 显示商品名称的行 -->
    <TableRow>
    <TextView
        android:text="@string/item_name"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/itemName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        style="@style/tv_show"/>
    </TableRow>
    <!-- 显示商品描述的行 -->
    <TableRow>
    <TextView
        android:text="@string/item_desc"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/itemDesc"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="@style/tv_show"/>
    </TableRow>
    <!-- 显示商品备注的行 -->
    <TableRow>
    <TextView
        android:text="@string/remark"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/itemRemark"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="@style/tv_show"/>
    </TableRow>
    <!-- 显示商品种类的行 -->
    <TableRow>
    <TextView
        android:text="@string/item_kind"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/itemKind"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="@style/tv_show"/>
    </TableRow>
    <!-- 显示起拍价格的行 -->
    <TableRow>
```

```
    <TextView
        android:text="@string/init_price"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/initPrice"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="@style/tv_show"/>
    </TableRow>
    <!-- 显示最高价格的行 -->
    <TableRow>
    <TextView
        android:text="@string/max_price"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/maxPrice"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="@style/tv_show"/>
    </TableRow>
    <!-- 显示结束时间的行 -->
    <TableRow>
    <TextView
        android:text="@string/end_time"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <TextView
        android:id="@+id/endTime"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="@style/tv_show"/>
    </TableRow>
    <!-- 输入竞拍价格的行 -->
    <TableRow>
    <TextView
        android:text="@string/you_bid"
        android:textSize="@dimen/label_font_size"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/bidPrice"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="numberDecimal"/>
    </TableRow>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center">
    <Button
        android:id="@+id/bnAdd"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/bid"/>
    <Button
        android:id="@+id/bnCancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/cancel"/>
    </LinearLayout>
    </TableLayout>
    </ScrollView>
```

当用户在竞拍界面中输入竞拍价格，单击“竞价”按钮后，程序将会向/Item/addBid 发送请求，添加竞价记录。

参与商品竞价的 Fragment 代码如下：

```
class AddBidFragment : Fragment() {
    // 定义界面中的文本框
    internal var itemName: TextView
    internal var itemDesc: TextView
    internal var itemRemark: TextView
    internal var itemKind: TextView
    internal var initPrice: TextView
    internal var maxPrice: TextView
    internal var endTime: TextView
    internal var bidPrice: EditText
    // 定义界面中的两个按钮
    internal var bnAdd: Button
    internal var bnCancel: Button
    // 定义当前正在拍卖的商品
```

```
        internal var jsonObj: JSONObject

        override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?, savedInstanceState: Bundle): View? {
            val rootView = inflater.inflate(R.layout.add_bid, container,
false)
            // 获取界面中的文本框
            itemName = rootView.findViewById(R.id.itemName) as TextView
            itemDesc = rootView.findViewById(R.id.itemDesc) as TextView
            itemRemark   =   rootView.findViewById(R.id.itemRemark)   as
TextView
            itemKind = rootView.findViewById(R.id.itemKind) as TextView
            initPrice   =   rootView.findViewById(R.id.initPrice)   as
TextView
            maxPrice = rootView.findViewById(R.id.maxPrice) as TextView
            endTime = rootView.findViewById(R.id.endTime) as TextView
            bidPrice = rootView.findViewById(R.id.bidPrice) as EditText
            // 获取界面中的两个按钮
            bnAdd = rootView.findViewById(R.id.bnAdd) as Button
                bnCancel = rootView.findViewById(R.id.bnCancel) as Button
            // 为取消按钮的单击事件绑定事件监听器
            bnCancel.setOnClickListener(HomeListener(activity))
            // 定义发送请求的 URL
            val url = HttpUtil.BASE_URL + "/Item/getItem?itemId="
            +arguments.getInt("itemId")
            try {
                // 获取指定的拍卖商品
                jsonObj = JSONObject(HttpUtil.getRequest(url))
                // 使用文本框来显示拍卖商品的详情
                itemName.text = jsonObj.getString("name")
                itemDesc.text = jsonObj.getString("desc")
                itemRemark.text = jsonObj.getString("remark")
                itemKind.text = jsonObj.getString("kind")
                initPrice.text = jsonObj.getString("initPrice")
                maxPrice.text = jsonObj.getString("maxPrice")
                endTime.text = jsonObj.getString("endTime")
            } catch (e1: Exception) {
                DialogUtil.showDialog(activity, "服务器响应出现异常！",
false)
                e1.printStackTrace()
            }

            bnAdd.setOnClickListener {
```

```
            try {
                // 执行类型转换
                val curPrice = java.lang.Double.parseDouble(
                        bidPrice.text.toString())
                // 执行输入校验
                if (curPrice < jsonObj.getDouble("maxPrice"))
                // ①
                {
                    DialogUtil.showDialog(activity,
                            "您输入的竞价必须高于当前竞价", false)
                } else {
                    // 添加竞价
                    val  result  =  addBid(jsonObj.getString("id"),
curPrice.toString() + "")  // ②
                    // 显示对话框
                    DialogUtil.showDialog(activity, result, true)
                }
            } catch (ne: NumberFormatException) {
                DialogUtil.showDialog(activity, "您输入的竞价必须是数值
", false)
            } catch (e: Exception) {
                e.printStackTrace()
                DialogUtil.showDialog(activity, "服务器响应出现异常，请
重试！", false)
            }
        }
        return rootView
    }

    @Throws(Exception::class)
    private fun addBid(itemId: String, bidPrice: String): String {
        // 使用 Map 封装请求参数
        val map = HashMap<String, String>()
        map.put("itemId", itemId)
        map.put("bidPrice", bidPrice)
        // 定义请求将会发送到/Item/addBid
        val url = HttpUtil.BASE_URL + "/Item/addBid"
        // 发送请求
        return HttpUtil.postRequest(url, map)
    }
}
```

从上面的程序可以看出，当用户单击“竞价”按钮后，程序将先执行输入校

验，然后再调用 addBid()方法来添加竞价记录。

如果用户竞价成功，系统将给出竞价成功提示信息。

7.9　本章小结

本章介绍了一个非常实用的 Android 应用，Android 应用充当电子拍卖系统的客户端，服务端采用 ThinkPHP+MySQL 的组合技术，系统架构采用的是模型-视图-控制器的分层结构，保证程序具有良好的可扩展性和可维护性。本系统的 Android 客户端通过 Apache HttpClient 与服务器交互，服务器与客户端采用的数据交换格式是 JSON。读者需要重点掌握 Android+PHP 的整合开发，以及通过网络进行数据交换的方式。